MASTER EN SCIENCES INFORMATIQUES

Logique et bases de données

Vanhentenryck, Pascal

Award date:
1985

Awarding institution:
Universite de Namur

LOGIQUE

ET

BASES DE DONNEES

Promoteur : Axel Van Lamsweerde

Pascal Vanhentenryck

année académique 84 - 85

ERRATA
LOGIQUE ET BASE DE DONNEES.

-pages 3 et 4 du plan inversées.
-pages 2 et 3 de l'introduction inversées.

Je tiens à remercier Monsieur Van Lamsweerde, promoteur de ce mémoire, pour avoir accepté de diriger ce travail, pour le stage qu'il m'a procuré et pour ses critiques lors de la rédaction de ce manuscrit.

J'exprime toute ma gratitude à Monsieur Yves Deville pour l'intérêt qu'il a porté à ce mémoire, ses critiques et ses réflexions ainsi que pour l'attention toute particulière qu'il a manifestée à mon égard.

Je tiens à exprimer, par ailleurs, ma reconnaissance à Monsieur Dincbas pour l'orientation qu'il a donnée à ce travail et la sympathie qu'il m'a témoignée.

Je remercie Monsieur Edouard André et toute l'équipe Concerto pour leur aimable accueil lors de mon stage à Lannion.

TABLE DES MATIERES

INTRODUCTION.

Ce mémoire concerne les relations qui existent entre la logique et les bases de données relationnelles ainsi que les principales contributions théoriques et pratiques de la logique à ces bases de données.

La logique formelle jouit déjà d'une très longue histoire. Elle fut introduite par Aristote, quatre siècles avant Jésus-Christ. Les logiques standards, la logique des propositions et la logique des prédicats du premier ordre, furent introduites à la fin du siècle dernier, notamment par Frege, Russel, Lowenheim, Gödel et Church. Le développement des logiques non standards (ie : la logique modale), a commencé au début du siècle avec C.I. Lewis tandis que Lukasiewiez et Post introduisaient les logiques à plusieurs valeurs de vérité. Les objectifs poursuivis par ces études furent très divers allant de la modélisation du raisonnement humain au fondement des mathématiques. Lorsque l'on s'intéresse à une logique, on définit son langage (qui définit les formules bien formées), sa sémantique (qui définit des conditions pour déterminer la valeur de vérité d'une formule) et sa théorie de la preuve (qui s'intéresse à développer des algorithmes pour déterminer la valeur de vérité d'une formule). L'informatique a ouvert de nouvelles voies à la logique : d'une part, le développement des procédures de preuve qui peuvent être exécutées efficacement sur ordinateur et d'autre part, son application à de nombreux domaines de l'informatique parmi lesquels on peut citer outre le champ "bases de données", la vérification de programmes, la synthèse de programmes, les systèmes questions-réponses, la résolution de problèmes et les langages de programmation.

Le modèle relationnel par Cood au début des années 70 a introduit une nouvelle approche dans le champ "bases de données".
Un modèle de données peut être considéré comme un ensemble de trois composants : un composant structuration qui définit les moyens d'expression dont on dispose pour représenter les informations, un composant manipulation qui définit un langage permettant de retrouver les informations mémorisées et un composant intégrité qui assure, dans la mesure du possible, la correspondance entre les informations mémorisées et la réalité modélisée. Parmi les principaux apports du modèle relationnel, on citera sa simplicité, son nombre réduit de concepts et sa séparation claire des niveaux logique et physique

d'une base de données. Le composant structuration permet d'exprimer les informations sans références explicites ou implicites à la manière dont elles sont mémorisées le composant manipulation permet de retrouver les informations sans références à la manière dont il faut les accéder et le composant intégrité permet d'exprimer des contraintes sur les informations modélisées sans références à la manière dont il faut les vérifier.

C'est à Green que l'on accorde généralement la première application de la logique formelle aux bases de données. Cependant, l'introduction du modèle relationnel fut déterminante pour la suite des rapports entre les deux champs. En effet, les concepts de relations et de prédicats présentent de nombreuses similitudes et il n'est guère étonnant que l'on ait pensé à utiliser le langage de logique des prédicats comme langage d'interrogation.
Nous verrons, cependant, que la contribution de la logique aux bases de données ne s'arrête pas à la seule utilisation de son langage.
En effet, une base de données relationnelle peut être formalisée en logique des prédicats et cette formalisation aura de nombreuses conséquences.

L'objectif de ce mémoire est de présenter les principales contributions de la logique aux bases de données relationnelles. Il sera structuré selon deux axes : l'axe logique
l'axe "composants du modèle relationnel".
L'axe logique consiste à séparer les apports des différentes logiques aux bases de données relationnelles.
L'axe "composants du modèle relationnel" consiste à séparer l'apport d'une certaine logique à chacun des composants du modèle relationnel.
De même, pour chacun des composants, nous trouverons toujours une section représentation et mise en oeuvre.

Comme il est d'usage dans un mémoire de réaliser une partie pratique nous avons tenté de mettre en application les concepts théoriques afin d'en étudier leur faisabilité. De nombreuses applications s'offraient à nous et nous en avons choisi un sous-ensemble représentatif étant donné le temps imparti pour ce mémoire. Toutes les applications ont été réalisées dans un langage de programmation logique. Examinons maintenant les contenus respectifs des différentes parties de ce travail.

La partie préliminaire est constituée de deux chapitres :
le premier concernant la logique des prédicats du premier ordre et

le second les bases de données relationnelles. Le premier chapitre présentera, dans un premier temps, la logique des prédicats du premier ordre à travers son langage, sa sémantique, sa théorie de la preuve et les résultats de complétude et de semi-décidabilité. Nous présenterons ensuite le principe de résolution qui est une règle d'inférence extrêmement efficace utilisé en démonstration automatique de théorèmes, en programmation logique et dans les bases de données déductives.
Nous terminerons ce chapitre par une présentation de la programmation en logique. Cette présentation se justifie, à la fois, parce que de nombreux concepts présentés dans ce mémoire peuvent être implémentés dans un langage de programmation logique et parce que cette étude a suscité chez nous un grand intérêt. Nous y présenterons les sémantiques opérationnelle et déclarative et un langage de programmation logique ainsi que la méthodologie de programmation de ce langage.
Nous terminerons par un certain nombre de mises en garde concernant la programmation en logique. Le chapitre "bases de données" s'attachera aux trois composants du modèle relationnel. Le composant structuration introduira les notions de relations d'attributs de domaines ...
Le composant manipulation présentera l'algèbre relationnelle qui peut être considérée comme le langage de référence pour les bases de données relationnelles. Enfin, le composant intégrité présentera les contraintes d'intégrité que devrait posséder toute base de données relationnelle, à savoir l'intégrité des identifiants et l'intégrité référentielle, ainsi que les notions de contrainte d'état, de contrainte de transition et de transaction.

La deuxième partie concernera la contribution de la logique des prédicats du premier ordre aux composants du modèle relationnel.
Elle se composera de quatre chapitres : le premier concerne la formalisation d'une base de données en logique des prédicats et les suivants la contribution de la logique à travers ces formulations aux trois composants du modèle relationnel.

Le premier chapitre "Formalisation d'une base de données relationnelles" mettra en évidence les hypothèses implicites prises en compte dans les bases de données relationnelles. Nous présenterons donc, l'hypothèse de fermeture du domaine, l'hypothèse du monde fermé et l'hypothèse d'unicité des noms. Nous définirons, ensuite, deux formalisations des bases de données relationnelles, respectivement la formalisation selon la vue "théorie du modèle" et la formalisation selon la

vue "théorie de la preuve".

Le deuxième chapitre "Bases de données déductives" concerne l'apport de la formalisation selon la vue "théorie de la preuve" au composant structuration. La section "représentation des informations" définira un nouveau moyen de représentation d'informations :les règles de déduction . Selon la forme de ces règles de déduction, on définira les bases de données déductives définies et indéfinies. La deuxième section "mise en oeuvre des règles de déductions" étudiera comment les bases de données déductives définies peuvent être implémentées. Nous étudierons de manière détaillée l'approche en génération. Nous avons réalisé un prototype de cette approche en programmation logique. Nous présenterons l'algorithme et tenterons d'en montrer la correction. Nous étudierons ensuite l'approche en dérivation.

Le troisième chapitre "Langages d'interrogation" concerne la contribution de la formalisation selon la vue "théorie du modèle" au composant manipulation. La section "représentation des questions" s'attachera à la modification de la notion d'interprétation pour définir la sémantique d'un langage d'interrogation. Cette modification pourra se faire de quatre manières distinctes selon que l'on considère les tuples ou les éléments du domaine comme objets primitifs et que l'on se place dans une logique typée ou non. Nous présenterons les langages d'interrogation basés sur une logique non typée. Nous discuterons, ensuite, de l'intérêt des types dans la base de données. Dans la section "mise en oeuvre des questions", nous étudierons les optimisations sémantiques basées sur la logique à savoir les optimisations syntaxiques et sémantiques.

Le quatrième chapitre "Contraintes d'intégrité" concerne la contribution de la formalisation selon la vue "théorie du modèle" au composant intégrité. La section "représentation des contraintes" se préoccuppera essentiellement de la représentation des contraintes de transition. La section "mise en oeuvre" présentera une méthode de simplification de contraintes d'intégrité que nous replacerons dans un cadre plus général. Nous discuterons ses avantages, ses inconvénients et ses extensions possibles sur base du cadre général et de l'expérience acquise lors de l'implémentation.

La troisième partie de ce mémoire concerne l'étude d'un certain

nombre de logiques non standards à savoir les logiques modales, les logiques à plusieurs valeurs de vérité et les logiques non monotones. Ces logiques concernent essentiellement des bases de données incomplètes où l'on remet en cause les hypothèses des bases de données relationnelles. Nous examinerons donc ces différentes logiques et montrerons quel est ou quel peut être leur impact sur le champ "bases de données".

Nous sommes bien conscients qu'une grande partie de ce mémoire a déjà été dit. Nous avons voulu, dans le cadre de ce travail, étudier, comprendre et réfléchir sur un certain nombre de concepts qui nous intéressaient. Notre point de départ fut la synthèse de Gallaire, Minker et Nicolas que nous avons voulu développer. Nous avons voulu l'étendre en considérant non seulement la logique des prédicats du premier ordre mais aussi d'autres logiques ainsi qu'en présentant la programmation en logique et ses liens avec les concepts présentés. De même, pour le chapitre "Base de données déductives", nous avons voulu présenter en détail l'approche en génération c-à-d sa spécification, la conception de l'algorithme et sa validation.

PARTIE I : PRELIMINAIRES.

CHAPITRE 1. LOGIQUE DES PREDICATS DU PREMIER ORDRE.

1.0 Introduction.

Dans ce chapitre, nous présenterons respectivement la logique des prédicats du premier ordre,le principe de résolution et la programmation en logique.
Dans un premier temps, on s'intéressera au langage, à la sémantique et à la théorie de la preuve de la logique des prédicats du premier ordre. Le langage définira l'ensemble des formules bien formées.
La sémantique s'attardera à déterminer la valeur de vérité des formules bien formées et on y introduira notamment les notions de modèles et d'interprétations.

La théorie de la preuve nous permettra de définir un système formel du calcul des prédicats qui nous donnera un moyen effectif de calculer la valeur de vérité des formules. Nous présenterons ensuite les résultats de complétude et de semi-décidabilité.
Dans un second temps, nous présenterons le principe de résolution qui est une règle d'inférence extrêmement efficace et qui est à la base de nombreux démonstrateurs automatiques de théorèmes. Il est également utilisé dans les interpréteurs de langages de programmation logique et pour mettre en oeuvre les bases de données déductives.
Enfin, nous présenterons la programmation en logique c.à.d. sa sémantique opérationnelle, sa sémantique déclarative, sa méthodologie de programmation ainsi qu'un certain nombre de critiques et de mises en garde.
Dans la sémantique opérationnelle, nous présenterons les résolutions linéaires, SL-résolution et SLD-résolution qui peuvent définir l'interpréteur d'un langage de programmation logique et la notion de contrôle.
La sémantique déclarative montrera comment un programme logique peut être utilisé pour calculer toute information qui est une conséquence logique du programme. Nous présenterons, ensuite, une méthodologie de programmation en logique qui consiste à séparer la partie logique de la partie contrôle. Nous terminerons par un ensemble de réflexions sur la programmation en logique.

1.1 La logique des prédicats du premier ordre : définitions.

Quand on s'intéresse à un système formel, on considère son langage, sa sémantique et sa théorie de la preuve. Le langage définira l'ensemble des formules bien formées, la sémantique s'attache à définir des conditions pour déterminer la valeur de vérité d'une formule et la théorie de la preuve s'efforce de trouver des moyens effectifs pour déterminer la valeur de vérité d'une formule bien formée. Nous examinerons respectivement chacun de ces aspects pour la logique des prédicats du premier ordre.

1.1.1 Définition du langage.

1.1.1.1 Alphabet d'un langage de premier ordre.

L'alphabet sera construit à partir des ensembles, finis ou récursifs et disjoints deux à deux suivants.

- X l'ensemble des variables individuelles
- C l'ensemble des constantes
- pour tout entier $n > 0$, F_n l'ensemble des symboles de fonction
- pour tout entier $n \geqslant 0$, P_n l'ensemble des symboles de prédicats n-aire

et à partir

- du connecteur logique '$\neg$' qui représente la négation logique
- du connecteur logique '$\rightarrow$' qui représente l'implication
- du quantificateur '$\forall$' qui représente le quantificateur universel.

Dans la suite, nous prendrons les conventions suivantes :

- les variables sont représentées par des lettres minuscules prises à la fin de l'alphabet.
- les constantes sont représentées par des lettres minuscules prises au début de l'alphabet .
- les symboles de prédicat sont représentés par des lettres majuscules.
- les symboles de fonction sont représentés par les lettres f,g,h.

1.1.1.2 Les formules bien formées.

Un terme est alors défini de la manière suivante :

(1) une constante est un terme .

(2) une variable est un terme .

(3) si f est un symbole de fonction n-aire et si $t_1, t_2, \ldots, t_n$ sont des termes, alors f $(t_1, t_2, \ldots, t_n)$ est un terme

(4) il n'y a pas d'autres termes.

Une formule atomique est définie de la manière suivante

Si P est un symbole de prédicat à n arguments ($n \geqslant 0$) et si $t_1, t_2, \ldots, t_n$ sont des termes, alors P $(t_1, t_2, \ldots, t_n)$ est une formule atomique. Dans le cas où n est égal à 0, nous dirons que P est une proposition.

Un littéral est une formule atomique (e.g. A) ou la négation d'une formule atomique (e.g. $\neg$ A).

Une formule bien formée peut être définie de la manière suivante

(1) une formule atomique est une formule bien formée.

(2) si A et B sont des formules bien formées, alors

(A) $\longrightarrow$ (B) est une formule bien formée.

A et B sont des sous-formules de la formule ainsi définie.

(3) si A est une formule bien formée,

$\neg$ (A) est une formule bien formée.

A est une sous-formule de la formule ainsi définie.

(4) si x est une variable et A une formule bien formée,

($\forall$ x) (A) est une formule bien formée.

A est une sous-formule de la formule ainsi définie.

Nous dénoterons wff l'appellation "formule bien formée".

Convention : nous noterons

(W1 $\wedge$ W2) la formule $\neg$ (W1 $\rightarrow \neg$ W2) où '$\wedge$' représente la conjonction.

(W1 $\vee$ W2) la formule ($\neg$ W1 $\rightarrow$ W2) où '$\vee$' représente la disjonction.

$(\exists x)$(Wl la formule $\neg((\forall x)\neg$ (Wl))où '$\exists$' représente le quantificateur existentiel.

De même, lorsqu'il n'y aura pas d'ambiguité, nous omettrons les parenthèses.

Variables libres et variables liées.

Soit x une variable. Soit A une formule bien formée. Nous dirons qu'une certaine occurence de x, dans A, est liée si et seulement si elle figure dans une sous-formule de A d'une des formes $(\forall x)A'$ ou $(\exists x)A'$. Sinon elle est libre.
Nous dirons qu'une variable x est liée dans A si A ne contient pas d'occurences libres de x. Sinon elle est libre.
Une formule bien formée dans laquelle toutes les variables sont liées est une formule bien formée fermée. Sinon c'est une formule bien formée ouverte.

Nous allons définir un certain nombre de formes particulières que peuvent prendre les wffs.

Forme normale prenexe conjonctive.

Une formule bien formée F est dite sous forme normale prenexe si et seulement si elle est de la forme

$$(q_1x_1) \ldots (q_nx_n)M$$

où $(q_ix_i)(1 \leqslant i \leqslant n)$ est soit $(\forall x_i)$ ou $(\exists x_i)$ et M est une wff qui ne contient pas de quantificateurs. $(q_1x_1) \ldots (q_nx_n)$ est appelé le préfixe et M est appelée la matrice de la wff F.
Elle est sous forme normale prenexe conjonctive si la matrice M est sous forme conjonctive c-à-d si M consiste en une conjonction de disjonctions de littéraux. Toute formule bien formée peut être mise sous forme normale prenexe conjonctive. Le lecteur intéressé par la transformation d'une formule bien formée quelconque en une formule bien formée sous forme normale prenexe conjonctive peut se reporter à [CHAN 73].

CHAN 73

Considérons la wff

$$\forall x\ \exists t(\exists y(P(x,y,a) \longrightarrow \exists z\ Q(y,z,t)) \wedge R(x,t))$$

Cette wff sous forme normale prenexe est donnée par

$$\forall x\ \exists t\ \forall y\ \exists z((P(x,y,a) \longrightarrow Q(y,z,t)) \wedge R(x,t)).$$

Sous forme normale prenexe conjonctive, elle se présente sous

la forme :

$$\forall x \exists t \forall y \exists z((\neg P(x,y,a) \vee Q(y,z,t)) \wedge R(x,t)).$$

Forme standard de Skolem.

Partant d'une wff sous forme normale prenexe conjonctive, les quantificateurs existentiels peuvent être éliminés en substituant aux variables qu'ils quantifient, des fonctions adéquates appelées fonctions de Skolem. La wff ainsi obtenue est dite sous forme standard de Skolem. La fonction de Skolem peut être déterminée de la manière suivante :

Soit une formule $(q_1x_1)\ldots(q_px_p)(\exists x)(q_{p+1}x_{p+1})\ldots(q_mx_m)M$.

La quantification $\exists x$ peut être supprimée en remplaçant toutes les occurences libres de x dans $(q_{p+1}x_{p+1})\ldots(q_mx_m)M$ par la fonction $f(x_{i_1},\ldots,x_{i_n})$ où

- f est un symbole de fonction n-aire distinct de tous les symboles de fonction existants.
- $x_{i_1},\ldots,x_{i_n}$ $(1 \leqslant i_j \leqslant p$ pour $j=1\ldots n)$ sont toutes les variables universellement quantifiées de $(q_1x_1)\ldots(q_px_p)$.

Si n = 0 alors nous obtenons une fonction de Skolem 0-aire c-à-d une constante de Skolem.

La wff précédente sous forme standard de Skolem devient :

$$\forall x \forall y((\neg P(x,y,a) \vee Q(y,f_1(x,y),f_2(x)) \wedge R(x,f_2(x)).$$

Le pas suivant dans ces normalisations successives nous amène à supprimer les quantificateurs universels en les rendant implicites. Cela nous amène au concept suivant.

Clauses.

Une clause est une disjonction de littéraux où les variables sont quantifiées universellement. Un ensemble de clauses est interprété comme la conjonction des clauses qu'il contient. C'est pourquoi la forme standard de Skolem n-aire donne lieu à l'ensemble de clauses

$$\{ P(x,y,a) \vee Q(y,f_1(x,y),f_2(x)), R(x,g(x))$$

Enfin une clause $\neg A_1 \vee \ldots \vee \neg A_n \vee B_1 \ldots \vee B_m$ peut s'écrire sous la forme

$$A_1 \wedge A_2 \wedge \ldots \wedge A_m \rightarrow B_1 \vee B_2 \vee \ldots \vee B_m$$

Dans le cas où n = 0 et m > 0, nous aurons :

$$\rightarrow B_1 \vee B_2 \vee \ldots \vee B_m$$

Si m = 0 et n > 0, nous aurons :

$$A_1 \wedge A_2 \wedge \ldots \wedge A_n \rightarrow$$

Nous prendrons, comme convention, que si m et n sont égaux à zéro, nous parlerons de la clause vide que nous noterons $\square$.

Une clause (resp. un littéral ou un terme) qui ne contient pas de variables est une clause (resp. un littéral ou un terme) de base. Par exemple, $P(a,f(a)) \vee S(a)$ est une clause de base. Une clause telle que $m \leqslant 1$ est une clause de Horn.

1.1.2 Sémantique : interprétations et modèles.

Soit L un langage de premier ordre.

Nous dirons qu'une interprétation de L consiste en

(1) la spécification d'un ensemble non vide d'éléments D, fini ou infini.

(2) une assignation qui

- à chaque constante c dans C associe un élément c' de D.
- à chaque symbole de fonction n-aire f de F_n, associe une fonction f' de D^n dans D.
- à chaque symbole de prédicat n-aire P, associe une fonction P' de D^n dans B où $B = \{ \text{vrai}, \text{faux} \}$.

La valeur de vérité d'une formule peut être calculée en définissant une fonction d'assignation g qui associe à chaque variable individuelle de X, un élément de D.

Par conséquent, la valeur de vérité d'une formule F dans une interprétation I étant donné une fonction d'assignation g, notée $\mathcal{S}_{I,g}(F)$, est définie par

(1) $\mathcal{S}_{I,g}(P(t_1,t_2,\ldots,t_n))$ est vrai si $P'(val(t_1,g),\ldots,val(t_n,g))$ est vrai
est faux sinon

où $val(t_i,g) = g(t_i)$ si t_i est une variable.

t_i' si t_i est une constante.

$f'(val(t_1',g),\ldots,val(t_m',g))$ si t est une fonction $f(t_1',t_2',\ldots,t_m')$.

(2) $\mathcal{S}_{I,g}(\neg A)$ est vrai si $\mathcal{S}_{I,g}(A)$ est faux.
est faux sinon

(3) $\mathcal{S}_{I,g}(A \rightarrow B)$ est vrai si $\mathcal{S}_{I,g}(\neg A)$ est vrai ou si $\mathcal{S}_{I,g}(B)$ est vrai

(4) $\mathcal{I}_{I,g}(\forall x\ A)$ est vrai si $\mathcal{I}_{I,g(d/x)}(A)$ est vrai pour tout $d \in D$
est faux sinon

où $g(d/x)$ est une fonction d'assignation identique à g sauf pour la variable x qui se voit assignée la valeur d.

Il est souvent plus commode de parler, par abus de langage, d'une interprétation d'un ensemble de formules bien formées S plutôt que de l'interprétation du langage de premier ordre dans lequel sont exprimées les formules. Nous supposons alors que le langage de premier ordre est par les constantes, les variables, les fonctions et les prédicats apparaissant dans S. Nous pouvons maintenant définir les notions suivantes.

- Une interprétation d'un ensemble de wffs est un modèle de cet ensemble si et seulement si toutes les wffs sont vraies dans cette interprétation.
- Une wff W est une conséquence logique d'un ensemble $\mathbb{E}$ de wffs si et seulement si W est vraie dans tous les modèles de $\mathbb{E}$. Nous noterons $\mathbb{E} \models W$ l'affirmation "W est une conséquence logique de $\mathbb{E}$ ".
- Un ensemblede wffs, $\mathbb{E}$, est satisfaisable ssi $\mathbb{E}$ a un modèle. Si $\mathbb{E}$ n'a pas de modèle, nous dirons que $\mathbb{E}$ est non-satisfaisable.
- Une wff W est valide si et seulement si elle est vraie dans toutes les interprétations possibles.
- Deux wffs sont équivalentes si elles prennent la même valeur de vérité dans toute interprétation. La forme normale prenexe conjonctive d'une wff W lui est équivalente. Par contre sa forme standard de Skolem ne l'est plus mais on peut montrer que si W est satisfaisable, alors sa forme standard l'est aussi.

1.1.3 Théorie de la preuve.

Un système formel est constitué d'un langage, d'un ensemble de wffs appelés axiomes et d'un ensemble de règles d'inférence ou de règles de déduction. Une règle d'inférence R est un algorithme admettant pour données $m_i + 1$ formules $g_1, g_2, \ldots, g_{m_i}, g$ $(m_i \geqslant 1)$ et fournissant pour résultat une des deux affirmations :

"g est déductible de $g_1, \ldots, g_{m_i}$ par la règle R"

ou

"g n'est pas déductible de $g_1, \ldots, g_{m_i}$ par la règle R".

Dans un système formel T, on appelle preuve, une suite finie non vide $f_1, f_2, \ldots, f_m$ de formules bien formées du langage du système formel, telle que pour chaque formule f_i, soit f_i est un axiome, soit f_i

est déductible de certaines formules $f_1, \ldots, f_{i-1}$ au moyen d'une règle d'inférence.

On appelle théorème du système formel, une formule qui figure dans une preuve. De même, une preuve $f_1, \ldots, f_m$ est une preuve de f si et seulement si $f = f_m$. Nous noterons $\vdash_T f$, l'affirmation "f est un théorème du système formel T ".

Si $f_1, f_2, \ldots, f_m$ sont des formules bien formées, nous noterons

$$f_1, f_2, \ldots, f_m \vdash_T f$$

l'affirmation

"f est un théorème du système formel T auquel on ajoute les axiomes $f_1, f_2, \ldots, f_m$".

Un système formel du calcul des prédicats est un système formel qui peut être défini par le langage que nous avons présenté précédemment, par un ensemble d'axiomes appelés axiomes logiques et par deux règles d'inférence.

1. Le modus ponens (ou syllogisme)

De (A) et (A) $\rightarrow$ (B) on infère (B)

2. La généralisation

Si P ne contient pas de variable libre x
De P on infère $(\forall x)P$.

Si nous ajoutons aux axiomes logiques, de nouveaux axiomes, on obtient un système formel appelé théorie du premier ordre. Ces nouveaux axiomes sont appelés axiomes propres ou axiomes non logiques.

Dans ce qui suit, nous noterons 'W$\vdash$F' l'affirmation 'F est un théorème d'un système formel du calcul des prédicats auquel on a ajouté les axiomes W'.

1.1.4 Résultat de complétude.

Dans la sémantique d'un système formel, on s'intéresse à la définition de conditions qui permettent de déterminer la valeur de vérité d'une formule. Dans la théorie de la preuve, on s'intéresse à la définition d'algorithmes qui permettent de déterminer la valeur de vérité d'une formule.

Le théorème de complétude Gödel nous indique que l'ensemble des

conséquences logiques d'un ensemble de wffs W est équivalent à l'ensemble des théorèmes du système formel qui aurait W comme axiomes propres. Ce résultat est vrai pour les théories du premier ordre mais ne l'est plus pour les théories d'ordre supérieur. Ce résultat comprend en fait deux notions :

(1) Complétude : Si une formule est vraie dans tous les modèles d'un ensemble W de formules bien formées, alors elle peut être prouvée dans une théorie du premier ordre qui contient W comme axiomes propres.

(2) Cohérence : Si une formule W est un théorème dans cette théorie du premier ordre, alors cette formule est vraie dans tous les modèles de W.

Ces résultats peuvent être schématisés par

$$W \models w \underset{\leftarrow\ \langle 2 \rangle}{\overset{\langle 1 \rangle}{\rightarrow}} W \vdash w$$

Considérons maintenant le problème suivant.

"Etant donnée une théorie du premier ordre et étant donné une wff w, existe-t-il une procédure de preuve générale qui nous dise si w est un théorème de cette théorie ?"

Church et Turing ont montré indépendamment l'un de l'autre qu'une telle procédure ne pouvait exister c-à-d qu'il n'existe pas une procédure de preuve générale qui étant donné une théorie du premier ordre et une formule bien formée w nous dise si w est un théorème ou non. Cependant, l'ensemble des théorèmes d'une théorie du premier ordre est énumérable. Par conséquent, au vu des résultats qui précèdent, le mieux que nous puissions attendre d'une procédure de preuve est qu'elle nous dise si une wff w est vraiment un théorème si cette wff en est effectivement un.

Si la wff n'est pas un théorème, alors la procédure de preuve refusera dans certains cas de répondre et bouclera indéfiniment.

Le problème est donc dit semi-décidable.

Afin d'éviter toute ambiguité, nous ferons la remarque suivante.

Considérons une procédure de preuve A qui nous dise qu'une wff est un théorème si cette wff en est effectivement un.

Il ne faudrait pas croire que l'on pourrait construire une procédure B qui répondrait systématiquement 'non' lorsque A ne répond pas.

En effet, ce qui précède signifie que la procédure A ne peut être

beaucoup plus intelligente qu'un algorithme qui appliquerait les règles de déduction, quelle que soit la sophistication du mécanisme qu'elle mettra en oeuvre. Ainsi, la procédure peut prévoir un certain nombre de cas particuliers mais dans la plupart des cas , elle n'aura d'autre recours que d'appliquer les règles de déduction.

Dès lors, si, à un instant donné, la procédure B prend l'initiative de suspendre A, rien ne lui garantit que, l'instant suivant, la procédure A n'aurait pas réussi à prouver que la wff est un théorème.

1.2. Le principe de résolution.

Plutôt que d'appliquer les règles d'inférence du modus ponens et de la généralisation pour déterminer les théorèmes d'une théorie du premier ordre, il est possible de construire des procédures de preuve qui soient basées sur des notions sémantiques telle que la satisfaisabilité et la non-satisfaisabilité.

Parmi les procédures de preuves les plus connues, se trouvent les procédures de preuve par réfutation conçues par Jacques Herbrand en 1930. Le principe sous-jacent à ces procédures de preuve est le suivant : afin de prouver qu'une wff est un théorème d'un ensemble d'axiomes W, nous montrerons que sa négation en conjonction avec l'ensemble des axiomes W est non satisfaisable.

Un progrès considérable pour les procédures de preuve par réfutation a été réalisé par Robinson [ROBI 65] qui a conçu le principe de résolution. Les procédures de preuve basées sur le principe de résolution sont beaucoup plus efficaces que les procédures conçues antérieurement. Ces procédures possèdent une seule règle d'inférence "Le principe de résolution" et Robinson a montré qu'une procédure de preuve basée sur le principe de résolution était cohérente et complète. C'est ce principe que nous allons présenter dans les pages qui suivent.

1.2.1 Cas de clauses de base.

Le principe de résolution s'applique à des formules bien formées mises sous forme de clauses. Dans un premier temps, et ce afin de faciliter la compréhension de l'exposé, nous présentons le cas où nous ne sommes en présence que de clauses de base. Le cas général sera présenté ensuite. Dans ce qui suit, nous considérerons qu'une clause est constituée d'un ensemble de littéraux.

définition : Si A est une formule atomique alors les deux littéraux A et ¬A sont dits être le complément l'un de l'autre et former une paire complémentaire. De même, on dira que A est un littéral positif et ¬ A un littéral négatif.

définition : Si C,D sont deux clauses de base et s'il existe une paire complémentaire de littéraux L dans C et ¬L dans D alors nous appellerons résolvante de base des clauses C et D, la disjonction de l'ensemble des littéraux formée par les littéraux de la clause C auxquels on a retranché L et les littéraux de la clause D auxquels on a enlevé ¬L.

Il découle de cette définition que deux clauses de base n'ont qu'un nombre fini de résolvantes puisque le nombre de littéraux est fini dans chacune des clauses de base. On remarquera aussi que tout modèle de $\{C,D\}$ est aussi un modèle de $\{C,D,E\}$ où E est une résolvante de C,D.

définition : Soit $\mathcal{S}$ un ensemble de clauses de base.
Nous noterons $\mathcal{R}(\mathcal{S})$, l'union de $\mathcal{S}$ et de toutes les résolvantes de base obtenues à partir des paires de membres de $\mathcal{S}$.
De la même manière, nous noterons

$$\mathcal{R}^{n+1}(\mathcal{S}) = \mathcal{R}(\mathcal{R}^{n}(\mathcal{S})).$$

Par conséquent, partant d'un ensemble fini de clauses de base, nous pouvons générer $\mathcal{R}(\mathcal{S})$, $\mathcal{R}^2(\mathcal{S})$ pour finalement obtenir $\mathcal{R}^n(\mathcal{S}) = \mathcal{R}^{n+1}(\mathcal{S})$ ce qui arrivera toujours puisqu'il n'existe qu'un nombre fini de clauses et de littéraux par clause. Nous aurons, alors, calculer la fermeture transitive de la relation $\mathcal{R}$.
D'autre part, si un ensemble de clauses de base est satisfaisable, alors nous pourrons trouver un littéral de base dans chaque clause sans être obligé de choisir une paire de littéraux complémentaires. Si un ensemble de clauses de base est non-satisfaisable, alors ce ne sera pas possible. On remarquera que la clause vide $\square$ est non-satisfaisable (puisque l'on ne peut pas choisir un littéral dans cette clause) et qu'un ensemble vide de clauses est satisfaisable.
Nous venons en fait d'énoncer le théorème de Robinson [ROBI 65]. De même, nous pouvons dire que tout ensemble de clauses contenant la clause vide est non-satisfaisable.

Théorème de Robinson (1) : Si $\mathcal{S}$ est une conjonction finie de clauses de base alors $\mathcal{S}$ est non-satisfaisable si et seulement s'il existe un $n \geqslant o$ tel que

$$\square \in \mathcal{R}^n(\mathcal{S})$$

Intuitivement ce théorème est justifiable de la manière suivante.

- Si $\square \notin \mathcal{R}^n(\mathcal{S})$ pour tout n, on a la relation en particuler pour n tel que $\mathcal{R}^n(\mathcal{S}) = \mathcal{R}^{n+1}(\mathcal{S})$ ce qui signifie que $\square$ n'appartient pas à la fermeture transitive de $\mathcal{R}$ pour $\mathcal{S}$. Cela veut donc dire que l'on a pu trouver un littéral dans chaque clause sans être obligé de prendre son complémentaire dans une autre clause et donc qu'il existe une interprétation dans laquelle toutes les clauses sont vraies c.à.d. un modèle de $\mathcal{S}$.

- Si $\square \in \mathcal{R}^n(\mathcal{S})$ comme tout modèle de $\mathcal{S}$ est aussi un modèle de $\mathcal{R}^n(\mathcal{S})$ et que $\mathcal{R}^n(\mathcal{S})$ n'a pas de modèle puisqu'il contient $\square$ alors $\mathcal{S}$ ne peut posséder de modèle.

Cela signifie qu'un ensemble de clauses de base $\mathcal{S}$ est non satisfaisable si et seulement si $\square \in \mathcal{R}^n(\mathcal{S})$ tel que $\mathcal{R}^n(\mathcal{S}) = \mathcal{R}^{n+1}(\mathcal{S})$.
Dès lors, pour tester la satisfaisabilité de $\mathcal{S}$, il suffit de partir de $\mathcal{S}$ et de générer $\mathcal{R}(\mathcal{S})$, $\mathcal{R}^2(\mathcal{S})$,... jusqu'à obtenir la clause vide ou la fermeture transitive de la relation.

1.2.2. Le principe de résolution.

---

définition : Soit $\mathcal{S}$ un ensemble de clauses.
Nous appellerons $\mathcal{H}$ l'univers d'Herbrand de $\mathcal{S}$ l'ensemble défini par les propriétés suivantes
(1) Si a est une constante dans $\mathcal{S}$ alors $a \in \mathcal{H}$.
(2) Si $a_1, \ldots, a_n$ sont des termes dans $\mathcal{H}$ et f est un symbole de fonction n-aire dans $\mathcal{S}$, alors $f(a_1, a_2, \ldots, a_n) \in \mathcal{H}$.
(3) Il n'y a pas d'autres éléments dans $\mathcal{H}$.

définition : Soit W une formule bien formée qui contient une variable v.
Soit t un terme.

Nous appellerons instance de W, la formule bien formée obtenue en substituant chacune des occurences de la variable v par le terme t. Cette instance est notée $W\{(t,v)\}$. De la même manière, si nous remplaçons toutes les occurences des variables $v_1, v_2, \ldots, v_n$ ($v_i \neq v_j$, $1 \leq i, j \leq n$ et $i \neq j$) respectivement par les termes $t_1, t_2, \ldots, t_n$ nous noterons $W\{(t_1,v_1), \ldots, (t_n,v_n)\}$.

définition : Une substitution σ est un ensemble ordonné de paires $\sigma = \{(t_1,v_1), \ldots, (t_n,v_n)\}$ le premier élément de chaque paire étant un terme et le second étant une variable telle que $v_i \neq v_j$ et $t_i \neq v_i$. Le résultat de l'application de σ à C, $C\sigma$, est l'instance $C\{(t_1,v_1),(t_2,v_2)$ $\ldots,(t_n,v_n)\}$ où C est une wff.

Nous noterons $P(\mathcal{S})$ l'ensemble des instances de clauses de base obte-

nues en appliquant à un ensemble $\mathcal{S}$ de clauses, toutes les substitutions possibles de termes appartenant à un ensemble P.

définition : On appelle expansion de Herbrand de $\mathcal{S}$, l'ensemble $\mathcal{H}(\mathcal{S})$.
On remarquera que cet ensemble sera infini
1) Si $\mathcal{S}$ contient au moins une variable.
et 2) Si $\mathcal{S}$ contient au moins un symbole de fonction n-aire avec $n > o$.

Théorème de Herbrand : Si $\mathcal{S}$ est une conjonction finie de clauses et $\mathcal{H}$ est l'expansion de Herbrand de $\mathcal{S}$ alors $\mathcal{S}$ est non-satisfaisable ssi il existe un sous-ensemble fini de $\mathcal{H}$ qui est non-satisfaisable.

Le théorème de Robinson permet de déterminer effectivement si un ensemble de clauses de base est satisfaisable. Le théorème d'Herbrand, quant à lui, nous apprend que, pour prouver qu'une conjonction de clauses (pas forcément de base) est non-satisfaisable, il suffit de prouver qu'un sous-ensemble de son expansion de Herbrand n'est pas satisfaisable.
Tout le problème va donc résider dans la recherche d'un sous-ensemble non-satisfaisable. Les premiers démonstrateurs de théorèmes utilisaient, pour ce faire, les niveaux de l'univers de Herbrand notés $\mathcal{H}_o, \mathcal{H}_1, \ldots \mathcal{H}_n, \ldots$ tels que
1) $\mathcal{H}_o$ est l'ensemble des constantes de $\mathcal{H}$.
2) $\mathcal{H}_{n+1}$ $(n \geq o)$ consiste en l'ensemble de tous les termes de $\mathcal{H}$ qui sont dans $\mathcal{H}_n$ ou dont les arguments sont dans $\mathcal{H}_n$.
Ces démonstrateurs généraient, dans un premier temps, toutes les clauses de base où les variables étaient remplacées par des constantes, puis toutes les clauses de base où les variables sont remplacées par des constantes et des fonctions qui n'ont que des constantes comme argument et ainsi de suite.
Ces démonstrateurs arrivaient rapidement à une croissance exponentielle des possibilités de substitutions en montant dans les niveaux de Herbrand.
Pour remédier à cette défaillance, Robinson a introduit l'idée suivante. Pourquoi, au lieu de générer toutes les instances possibles de l'expansion de Herbrand, ne pourrait-on pas essayer de prédire quels sont les sous-ensembles qui vont s'avérer incohérents.
Pour illustrer cette idée, considérons l'exemple suivant.

Certains concepts, mentionnés dans l'exemple, seront définis par la suite.

Soit l'ensemble de clauses

(1) $P(x,d,x)$.

(2) $\neg P(y,v,w,) \vee \neg P(y,z,v) \vee P(d,z,w)$

(3) $P(a,f(u,v),d)$

(4) $\neg P(d,f(f(b,c),a),a)$.

où a,b,c,d sont des constantes.

La clause (1) et la clause (2) admettent comme résolvante

(5) $\neg P(x,z,d) \vee P(d,z,x)$

car une instance de (2) est

$\neg P(x,d,x) \vee \neg P(x,z,d) \vee P(d,z,w)$.

Cette résolvante (5) est en fait une famille de résolvantes puisqu'elle contient des variables.

La résolvante (5) et la clause (3) donnent lieu à la nouvelle famille de résolvantes (6) $P(d,f(u,v),a)$.

Cette famille contient l'instance $P(d,f(f(b,c),a),a)$ et cette instance contredit manifestement la clause (4). Nous avons pu trouver un sous-ensemble de l'expansion de Herbrand qui est non-satisfaisable et donc l'ensemble des clauses (1) - (4) est non satisfaisable. En l'occurrence, ce sous-ensemble non-satisfaisable est

(1) $P(a,d,a)$

(2) $\neg P(a,d,a) \vee \neg P(a,f(f(b,c),a), \vee P(d,f(f(b,c)),a))$

(3) $P(a,f(f(b,c),a),d)$

(4) $\neg P(d,f(f(b,c),a),a)$.

Il devrait être clair que les premiers démonstrateurs de théorèmes auraient dû générer de très nombreuses clauses de base avant de trouver ce sous-ensemble.

Pour remédier à cet inconvénient, on a donc pensé au mécanisme suivant : pour deux clauses, on essaie de déterminer une substitution de termes (pas nécessairement de base) pour les variables de deux clauses de telle manière que le résultat de cette substitution produise un littéral L dans une clause et un littéral $\neg$ L dans l'autre.

Composition de substitutions.

Soit Ψ et θ deux substitutions.

La composition de θ et de Ψ notée $\theta\Psi$ est la substitution définie par

Soit $\theta = (t_1,u_1), \ldots, (t_n,u_n)$,

soit $\theta' = (t_1\Psi, u_1), \ldots, (t_n\Psi, u_n)$ tel que $t_i \Psi \neq u_i$

et Soit Ψ' l'ensemble des paires de Ψ dont le second élément n'apparait pas parmi les variables $u_1, \ldots, u_n$ de θ.

$$\Psi' = \{(s_1, v_1), \ldots (s_k, v_k)\}.$$

alors $\theta\Psi = \theta' \cup \Psi'$

On peut montrer que le résultat de l'application de θ et Ψ successivement à une formule bien formée C est identique à l'application de $\theta\Psi$ à C

$$(C\,\theta)\,\Psi = C\,(\theta\Psi)$$

De plus, la composition de substitutions est associative

$$(\theta\sigma)\Psi = \theta(\sigma\Psi).$$

définition : soit $\mathcal{L} = \{L_1, L_2, \ldots, L_n\}$ un ensemble de littéraux. $\mathcal{L}$ est unifiée (au sens de Robinson) par σ si et seulement si

$$L_1\sigma = L_2\sigma = \ldots = L_n\sigma.$$

On remarquera aisément que si σ unifie $\mathcal{L}$ alors toute substitution $\sigma\lambda$ unifie $\mathcal{L}$ où λ est toute substitution.

définition : soit $\mathcal{L} = \{L_1, L_2, \ldots, L_n\}$ un ensemble de littéraux. Nous dirons que θ est la substitution la plus générale qui unifie $\mathcal{L}$ si pour toute substitution σ qui unifie $\mathcal{L}$, il existe une substitution λ telle que $\sigma = \theta\lambda$.

Remarquons qu'il n'est pas toujours possible d'unifier un ensemble de littéraux mais lorsque cet ensemble s'unifie, il est toujours possible de trouver la substitution la plus générale et de plus, il existe un algorithme pour le calculer.

Nous allons maintenant donner la définition de résolvante dans le cas général pour ensuite donner l'énoncé du principe de résolution.

définition : une résolvante de deux clauses C_1 et C_2 est une troisième clause D obtenue de la manière suivante

1) soit $v_1, v_2, \ldots, v_n$ les variables de C_2. Nous pouvons définir une substitution $\theta = \{(u_1, v_1), \ldots, (u_n, v_n)\}$ de telle manière que C_1 et C_2 n'aient plus de variables en commun. Cette opération est appelée le renommage de variables.

2) soit deux ensembles de littéraux $\mathcal{L} = \{L_1, \ldots, L_k\}$ et

$M = \{M_1, \ldots, M_n\}$ tels que $\mathcal{L} \subseteq C_1$, $M \subseteq C_2$ et l'ensemble $\{l_1, \ldots, l_k, \neg M_1\theta, \ldots, \neg M_n\theta\}$ est unifiable.
Soit σ_o la substitution la plus générale telle que $\mathcal{L}\sigma_o$ et $M\theta\sigma_o$ soient des littéraux complémentaires

alors D est la clause formée de $(C_1 \setminus \mathcal{L})\sigma_o \cup (C_2 \setminus M)\theta\sigma_o$.

On remarquera que toute paire de clauses possède un nombre fini de résolvantes car il existe seulement un nombre fini de paires d'ensemble de littéraux $\mathcal{L}$ et M et que pour chaque paire, il existe au plus une substitution σ_o.
Nous noterons aussi que les résolvantes d'une paire de clauses de base sont des résolvantes de base.

Principe de résolution. Pour toutes deux clauses C_1, C_2, on infère une résolvante de C_1, C_2.

Une réfutation d'un ensemble de clauses $\mathcal{S}$ est une séquence $C_1, C_2, \ldots, C_n$ de clauses telle que chaque clause C_i appartient à $\mathcal{S}$ ou est inférée à partir de deux clauses de $C_1, \ldots, C_{i-1}$ et telle que $C_n = \square$.

Comme précédemment, nous savons que si $\mathcal{R}(\mathcal{S})$ désigne toutes les clauses de $\mathcal{S}$ et toutes les résolvantes de paires de clauses dans $\mathcal{S}$, $\mathcal{R}^{n+1} = \mathcal{R}(\mathcal{R}^n(\mathcal{S}))$. C'est pourquoi, il existe une réfutation de $\mathcal{S}$ ssi $\square \in \mathcal{R}^n(\mathcal{S})$ pour un certain $n \geq o$.

Nous venons de présenter le principe de résolution qui est une règle d'inférence efficace pour générer de nouvelles clauses à partir d'anciennes. Cependant, une application aveugle de ce principe conduirait à la génération de nombreuses clauses qui ne nous seraient d'aucune utilité et qui, par conséquent, nuiraient à l'efficacité.

Considérons l'exemple vu précédemment. Ajoutons les clauses

$$q(x,d,x)$$
$$\neg q(x,y,z) \vee q(a,d,b)$$

L'application du principe de résolution à ces deux clauses n'est guère utile pour prouver que l'ensemble est non-satisfaisable.

Dans le but d'obtenir des démonstrateurs de théorèmes efficaces, nous devrons nous efforcer d'empêcher, autant que possible, la génération de ces clauses sans utilité.
C'est pourquoi le principe de résolution a fait l'objet d'un grand nombre de raffinements. Chang & Lee identifient la sémantic-résolution, la lock-résolution et les résolutions linéaires comme les trois raffinements les plus importants. Nous aurons l'occasion de revenir sur les stratégies linéaires ultérieurement.
Le lecteur intéressé par les différentes mises en oeuvre de principe de résolutionpeut se reporter à [CHAN 73].

1.3 Programmation en logique.

1.3.1 Historique.

L'utilisation de la logique pour définir un langage de programmation trouve son origine dans des recherches effectuées en démonstration automatique de théorèmes. Aux alentours des années 60, un certain nombre de démonstrateurs de théorèmes furent conçus mais l'explosion exponentielle des combinaisons nécessaires pour générer une preuve, les rendaient pratiquement inutilisables dès que le problème à résoudre sortait du cadre des problèmes trivialement simples.

Un progrès substantiel et décisif fut réalisé par la définition du principe de résolution [ROBI 65] que nous venons de présenter. L'idée sous-jacente était extrèmement simple lorsque nous la considérons avec un recul de 20 ans. Au lieu d'essayer aveuglément toutes les combinaisons possibles d'instantiations sur l'univers d'Herbrand, pourquoi ne pourrait-on pas prédire les instantiations qui conduiraient à la "combinaison gagnante", autrement dit à la combinaison qui permettrait de générer une preuve ?

Le principe de résolution fut alors exploité dans les démonstrateurs de théorèmes tandis que, simultanément, des recherches sur les stratégies efficaces pour le mettre en oeuvre étaient entreprises. Ces recherches aboutirent notamment à la conception d'une stratégie extrêmement efficace : SL-résolution [KOWA 71].
Cette stratégie, lorsque nous nous restreignons au sous-ensemble de la logique constitué des clauses de Horn ressemble étrangement à un interpréteur de langage de programmation conventionnel.
Par ailleurs, le travail de Boyer et Moore [BOYE 72] mettait en évidence que les techniques d'implémentation efficace d'un démonstrateur de théorèmes n'étaient pas tellement éloignées de l'implémentation efficace d'un langage de programmation.
Parallèlement à ces études, Green [GREE 69] montrait comment un système de questions-réponses pouvait être construit en utilisant la logique du premier ordre comme langage et un démonstrateur de théorèmes basé sur le principe de résolution comme mécanisme de déduction. En outre, son article montrait comment un démonstrateur de théorème pouvait "simuler" un calcul. Nous employons le terme "simuler" car l'objectif du système de Green n'était pas de réaliser un calcul avec démonstrateur de théorèmes et par conséquent, si ce système pouvait effectivement réaliser un calcul (en l'occurence, la recherche d'un chemin dans un graphe), c'était de manière tout à fait inefficace.

Ce sont là les principales contributions qui ont amené Hayes [HAYE 73] et Kowalski [KOWA 74] à proposer de considérer le calcul comme une déduction contrôlée. Hayes définissait ses programmes comme un ensemble d' équations tandis que Kowalski utilisait la logique du premier ordre restreinte aux clauses de Horn.

L'implémentation du langage de programmation Prolog [COLM 73] constitue une dernière étape qui a conduit à l'utilisation de la logique comme langage de programmation. A ce jour, nombre d'implémentations ont été réalisées et Warren [WARR 77] a montré que pour des problèmes de manipulation de listes, les programmes prolog pouvaient être favorablement comparés avec des programmes compilés en Lisp.

1.3.2. Sémantique opérationnelle des programmes logiques.

Pour décrire la sémantique opérationnelle d'un programme logique, nous allons donner la définition d'un interpréteur sur une machine abstraite. Tout démonstrateur de théorème basé sur le principe de résolution peut être utilisé comme interpréteur d'un langage logique. Cependant, pour d'évidentes raisons d'efficacité et de facilité de mise en oeuvre, les interpréteurs limitent l'expression des programmes aux clauses de Horn.
Au moment où Prolog a été conçu, on pouvait le considérer comme une mise en oeuvre de SL- résolution [KOWA 71] avec un certain nombre de restrictions et de choix. De nos jours, les langages de programmation logique sont plutôt considérés comme une mise en oeuvre de Lush-résolution ou SLD-résolution. [HILL 75], [LLOY 84]. Comme SL-résolution et SLD-résolution sont des résolutions linéaires, nous allons présenter dans un premier temps les résolutions linéaires. Nous présenterons brièvement SL-résolution pour ensuite nous attarder un peu plus sur SLD-résolution. Nous terminerons par une présentation de la sémantique opérationnelle d'un langage de programmation logique et des choix particuliers implémentés en Prolog.

1.3.2.1 Les résolutions linéaires.

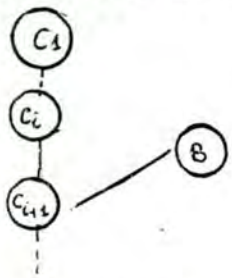
Les résolutions linéaires vont définir des conditions sur la manière dont on peut appliquer le principe de résolution. Schématiquement, partant d'un ensemble de clauses, on en prendra une pour laquelle on appliquera le principe de résolution donnant lieu à une résolvante. Cette résolvante sera nécessairement une des clauses de la prochaine

application du principe de résolution et ainsi de suite.

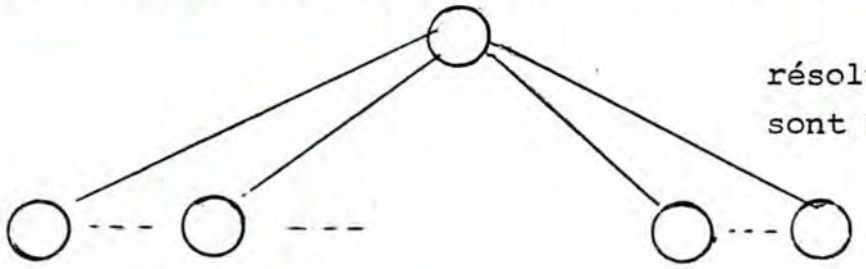
Soit S un ensemble de clauses. Une dérivation linéaire de S est une suite de clauses $C_1, C_2, \ldots, C_n$ telle que

- $C_1 \in S$
- C_{i+1} est une résolvante de C_i d'une clause B qui est
  - soit dans S auquel cas C_{i+1} est dit obtenu par résolution par extension
  - soit C_j avec $j < i$ auquel cas C_{i+1} est dit obtenu par résolution par ancêtre.

Une réfutation linéaire de S est une dérivation linéaire de S qui a la clause vide comme dernière clause.



La résolution linéaire définit un espace de recherche de toutes les dérivations possibles. Cet espace peut être représenté par un arbre de recherche où la racine est une clause de S. Chaque résolvante possible représente un noeud. Un noeud admet autant de descendants qu'il existe de résolvantes de ce noeud obtenues par résolution par extension ou par résolution par ancêtre. Etant donné une résolvante, n'importe quel littéral pourra être choisi pour être résolu c.à.d. pour tenter de l'unifier avec un littéral d'une clause B définie comme ci-dessus. De même, il peut y avoir o,1 ou plusieurs manières de résoudre un littéral. Par conséquent, les résolvantes obtenues par résolution à partir d'une résolvante donnée peuvent être schématisées par



résolvante dont les littéraux sont $L_1, L_2, \ldots, L_n$

résolvante dérivée de la résolution de L_1

résolvante dérivée de la résolution de L_n

On peut montrer que les résolutions linéaires conservent la complétude et la cohérence.

1.3.2.2 SL-résolution [KOWA 71] .

SL-résolution (linear resolution with selection function) est une résolution linéaire qui impose une restriction sur le littéral qui va être choisi pour la résolution.
Si, à un instant donné, nous avons une résolvante

$$B_1 \vee B_2 \vee \ldots \vee B_m$$

et que le littéral B_i est choisi pour être résolu et que nous supposons que B_i est résolu par extension avec une clause

$$B \vee A_1 \vee \ldots \vee A_n .$$

La résolution donne lieu à la clause

$$(B_1 \vee B_2 \vee \ldots \vee B_{i-1} \vee A_1 \vee \ldots \vee A_n \vee B_{i+1} \vee \ldots \vee B_m)\,\theta$$

où θ est la substitution la plus générale telle que B_i et B s'unifient.
SL-résolution impose la contrainte que nous ne pourrons résoudre $B_1, \ldots, B_{i-1}, B_{i+1}, \ldots B_m$ qu'après avoir résolu $A_1, \ldots, A_n$.
Cela signifie que les littéraux introduits par la résolution sont résolus sur une base last-in, first-out.
On a montré que SL-résolution est complète et cohérente.

1.3.2.3 SL-résolution [HILL 75] [LLOY 84] .

La plupart des langages de programmation logique peuvent être considérés comme une mise en oeuvre de SLD-résolution (ou Lush-résolution). C'est pourquoi, nous allons nous y attarder quelque peu.

définition : une expression est soit un terme soit un littéral soit une conjonction de littéraux ou une disjonction de littéraux. Soit E et F deux expressions. Nous dirons que E est une variante de F et réciproquement s'il existe deux substitutions θ et σ telles que $E = F\theta$ et $F = E\sigma$.

exemple : P (f(x,y), g(z), a) est une variante de P (f(y,x), g(u), a)
P (x,x) n'est pas une variante de P (x,y).

définition : une clause définie est une clause qui contient exactement un littéral positif.

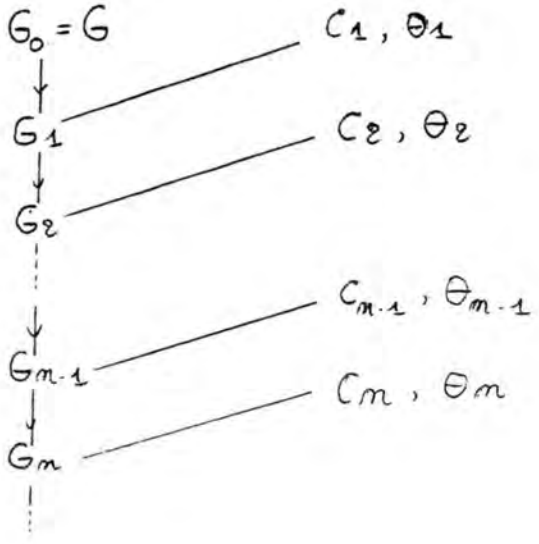
définition : une question est une clause qui ne contient que des littéraux négatifs.

définition : une fonction de sélection est une fonction d'un ensemble de questions vers un ensemble de formules atomiques telle que la valeur de cette fonction pour une question est toujours une formule atomique appelée le littéral choisi dans cette question.

définition : soit une question $\leftarrow A_1 \wedge A_2 \wedge \ldots \wedge A_k$ une question notée G.
soit C une clause $A \leftarrow B_1 \wedge \ldots \wedge Bq$.
soit R une fonction de sélection.
G-D est dérivée de G et de C en utilisant la substitution la plus générale θ via R si les conditions suivantes sont vérifiées :
1° A_m est le littéral choisi par la fonction de sélection R.
2° $A_m \theta = A \theta$.
3° G-D est la question
$\leftarrow (A_1 \wedge \ldots \wedge A_{m-1} \wedge B_1 \wedge \ldots \wedge B_q \wedge A_{m+1} \wedge \ldots \wedge A_k)$.

définition : soit P un ensemble de clauses définies,
soit G une question,
soit R une fonction de sélection,
Une dérivation-SLD de $P \cup \{G\}$ consiste en une séquence finie ou infinie $G_0 = G, G_1, \ldots$ de questions, une séquence $C_1, C_2, \ldots$ de variantes de clauses de P et une séquence $\theta_1, \theta_2, \ldots$ de substitutions les plus générales telles que chaque G_{i+1} est dérivé de G_i et C_{i+1} en utilisant θ_{i+1} via R.

On notera que C_i est une variante des clauses de P telle que C_i ne contienne aucune variable qui soit apparue dans la dérivation réalisée jusque G_{i-1}



définition : une SLD-réfutation de $P \cup \{G\}$ via R est une SLD-dérivation de $P \cup \{G\}$ via R qui a la clause vide comme dernière question de la dérivation.

Les SLD-dérivations peuvent être finies ou infinies. Une dérivation finie peut être une dérivation avec succès ou une dérivation avec échec. Une SLD-dérivation avec succès est une SLD-dérivation qui se termine avec la clause vide ou encore une SLD-réfutation. Une SLD-dérivation avec échec est une SLD-dérivation qui se termine avec une question non vide et telle que le littéral choisi dans cette question ne s'unifie avec aucun littéral positif dans P.

Nous remarquerons que SLD-résolution ne met en oeuvre que la résolution par extension. Cela à plusieurs conséquences.
D'une part, elle ne sera plus complète pour des clauses quelconques et devra se restreindre à des clauses de Horn, pour lesquelles on a montré qu'elle était complète et cohérente.
D'autre part, cette restriction lui permet sa simplicité et son efficacité de mise en oeuvre. De plus, on notera que ce résultat est vrai, indépendamment du choix de la fonction de sélection du littéral laissant ainsi à une implémentation particulière le choix de cette fonction

SLD-résolution définit également un espace de recherche de toutes les dérivations possibles. Nous pouvons définir cet espace pour une fonction de sélection fixée.

définition : soit S un ensemble de clauses définies. Soit G une question et R une fonction de sélection. Un arbre-SLD pour $P \cup \{G\}$ via R est défini par

(1) Chaque noeud de l'arbre est une question (éventuellement vide).
(2) La racine est G.
(3) Soit $\leftarrow A_1 \wedge \ldots \wedge A_m \wedge \ldots \wedge A_k$ $(k \geq 1)$ un noeud de l'arbre et soit A_m le littéral choisi. Ce noeud a un descendant pour chaque clause $A \leftarrow B_1 \wedge \ldots \wedge Bq$ telle que A_m et A s'unifient. Le descendant est $\leftarrow (A_1 \wedge \ldots \wedge A_{m-1} \wedge B_1 \wedge \ldots B_q \wedge A_{m+1} \wedge \ldots \wedge A_k)\theta$.

Considérons les clauses

$$P(x,z) \leftarrow Q(x,y) \wedge P(x,z)$$
$$P(x,x) \leftarrow$$
$$Q(a,b) \leftarrow$$

où a et b sont des constantes.

Soit la question $\leftarrow P(x,b)$

les deux figures nous donnent deux exemples d'arbre-SLD. Le premier adopte comme fonction de sélection le littéral le plus à gauche. Le second, le littéral le plus à droite.

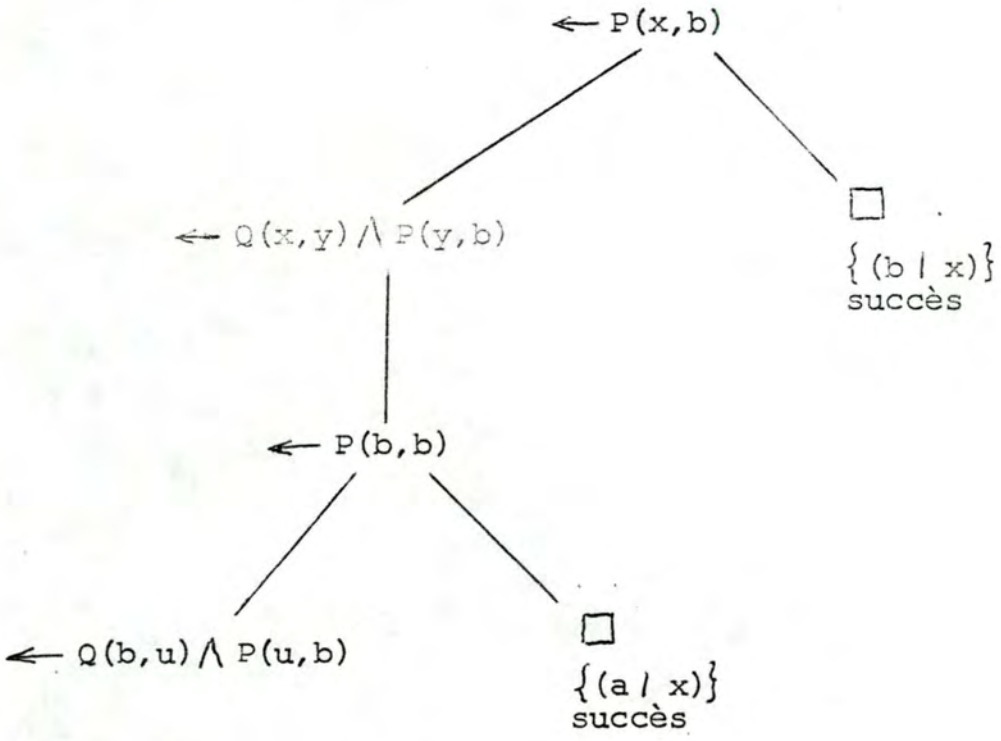


Figure 1.

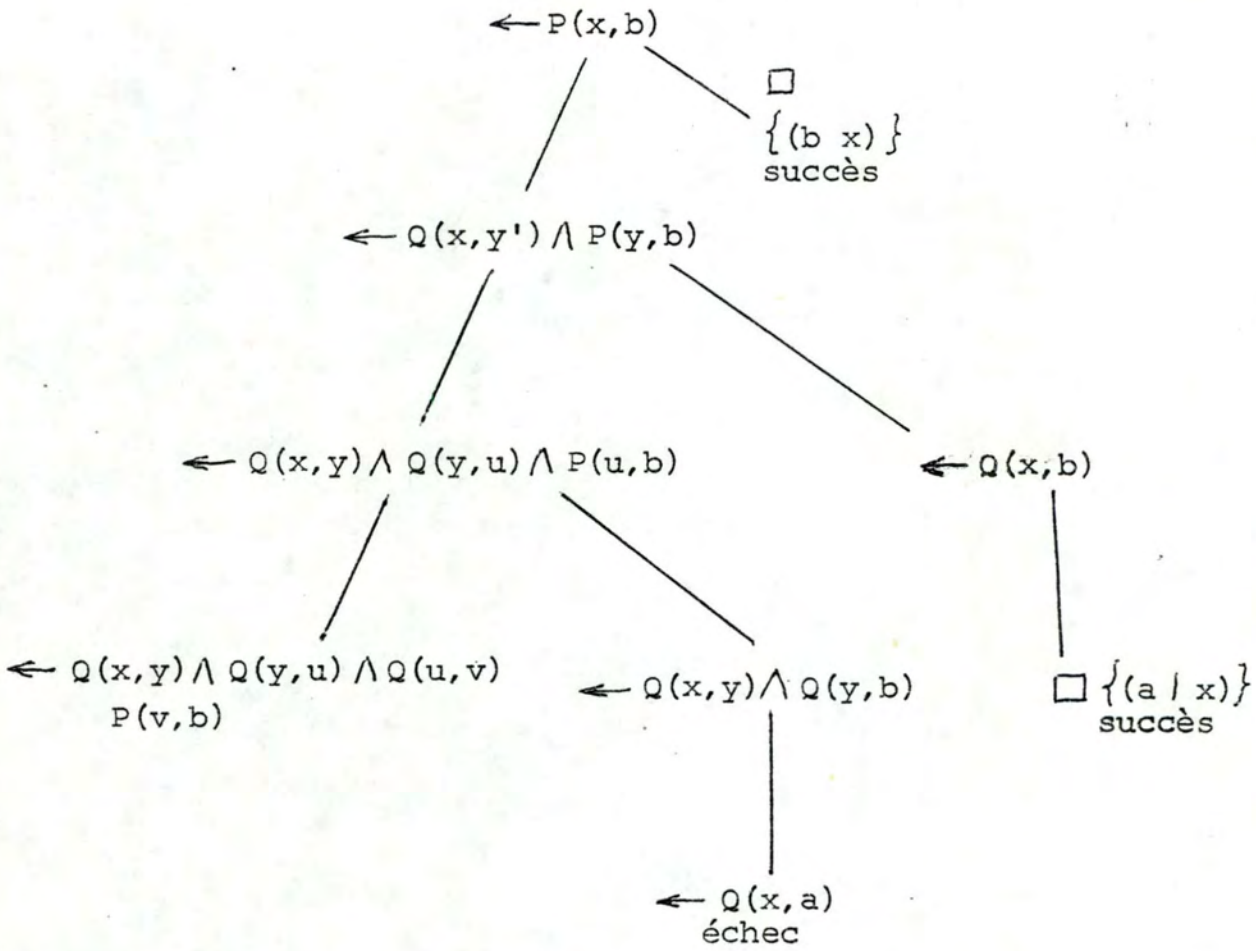

← P(x,b)
□
{(b x)}
succès
← Q(x,y') ∧ P(y,b)
← Q(x,y) ∧ Q(y,u) ∧ P(u,b)
← Q(x,b)
← Q(x,y) ∧ Q(y,u) ∧ Q(u,v)
P(v,b)
← Q(x,y) ∧ Q(y,b)
□ {(a / x)}
succès
← Q(x,a)
échec

Figure 2.

où θ est la substitution la plus générale telle que A_m et A s'unifient.

(4) Les noeuds qui sont des clauses vides n'ont pas de descendants.

Chaque branche de l'arbre-SLD est une dérivation de $P \cup \{G\}$. Les branches correspondant à des dérivations infinies sont des branches infinies et les branches correspondant à des dérivations avec échec sont appelées "branches avec échec".

1.3.2.4 Sémantique opérationnelle d'un langage de programmation logique.

L'interpréteur d'un langage de programmation logique n'est rien d'autre qu'une mise en oeuvre de SLD-résolution. Une fois que l'on a choisi une fonction de sélection pour SLD-résolution, nous avons défini un arbre de recherche. L'interpréteur devra alors choisir une stratégie d'exploration de cet arbre. Pratiquement, cela se manifeste de la manière suivante :

Lors de la résolution d'une question

$$\leftarrow A_1 \wedge \ldots \wedge A_m \wedge \ldots \wedge A_k$$

avec une fonction de sélection qui détermine A_m comme littéral choisi, il peut y avoir plusieurs clauses $C \leftarrow B_1 \wedge \ldots \wedge B_q$ telles que A_m et C s'unifient.

Chacune de ces clauses peut être choisie.

Comme ce choix peut nous aiguiller vers des branches avec échec alors qu'il existe des branches avec succès, il sera nécessaire d'élaborer une stratégie de remise en cause des choix.

Nous pouvons faire ici un rapprochement avec le problème de l'analyse syntaxique en théorie de la compilation où un même symbole terminal peut être résolu par plusieurs règles de production.

Nous allons maintenant définir ce qu'est un programme logique et comment nous pouvons l'utiliser.

définition : un programme logique est un ensemble de clauses définies.

définition : soit P un programme logique, soit Q une question, soit R une fonction de sélection.
Une substitution-réponse-via-R pour $P \cup \{Q\}$ est la substi-

tution obtenue en restreignant la composition $\theta_1\theta_2\ldots\theta_m$ aux variables de Q où $\theta_1\theta_2\ldots\theta_m$ est la séquence des substitutions les plus générales utilisées dans une SLD-réfutation de $P\cup\{Q\}$ via R.

Considérons maintenant un programme logique P et une question Q. La question Q doit être considérée comme une demande de recherche d'une SLD-réfutation du programme P avec la question q via la fonction de sélection de l'interpréteur.

Le résultat de cette demande est le suivant

Si l'interpréteur a trouvé une SLD-réfutation et que la question contient des variables
le résultat est la substitution-réponse-via-R de $P\cup\{Q\}$ si R est la fonction de sélection de l'interpréteur.

Si l'interpréteur a trouvé une SLD-réfutation et que la question ne contient pas de variables
le résultat est "oui" où "oui" est une convention pour la substitution vide.

Si l'interpréteur n'a trouvé aucune SLD-réfutation alors la réponse est "non".

Nous pouvons également donner une interprétation procédurale des clauses de Horn. Une clause de Horn du type implication ou assertion

$$A \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_n$$

peut être interprétée comme une déclaration de procédure. A détermine la forme des problèmes que peut résoudre cette procédure et est appelé le nom de la procédure. $B_1 \wedge B_2 \wedge \ldots \wedge B_n$ est appelé le corps de la procédure et est constitué d'un ensemble d'appels de procédures.
Une clause de Horn du type question

$$\leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_n$$

est interprétée comme une demande d'exécution des procédures $B_1, \ldots, B_n$.

1.3.2.5 Le composant de contrôle.

Dans les pages qui précèdent, nous avons mentionné un certain nombre de choix qui doivent être effectués lors de la conception d'un langage de programmation logique. Ces choix portent essentiellement sur

- la détermination de la fonction de sélection d'un littéral dans une question
- la détermination d'une règle de recherche qui est une stratégie de parcours des arbres-SLD afin de trouver des branches avec succès.

Le langage de programmation logique va prendre la responsabilité d'un certains nombres de choix sans laisser l'occasion au programmeur d'agir sur ces choix. Il va également permettre à l'utilisateur du langage d'exprimer le contrôle de l'exécution. Il existe différents moyens pour exprimer le contrôle et nous les évoquerons brièvement par la suite.

1.3.2.6 Prolog : un langage de programmation logique.

Présenter Prolog maintenant revient à présenter les choix de contrôle qui y sont effectués.

Prolog est une mise en oeuvre de SLD-résolution où la fonction de sélection choisit le littéral le plus à gauche dans la question et où la règle de recherche est un parcours de l'arbre-SLD en profondeur d'abord. De manière analogue, nous pouvons dire que Prolog implémente SLD-résolution où le choix du littéral se fait de gauche à droite dans la question où la remise en cause des échecs se fait par backtracking chronologique (c.à.d. que c'est toujours le dernier choix qui est remis en cause) et où le choix des clauses s'effectue dans l'ordre d'écriture du programme. Considérons brièvement ces choix.

Le choix du littéral de gauche à droite dans la question impose que si $B_1, B_2, \ldots, B_m$ sont les dernières formules atomiques entrées dans la question, et si $B_1, B_2, \ldots B_m$ proviennent d'une clause où elles étaient ordonnées selon l'ordre des indices,

$$A \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_m$$

Prolog résoudra toujours B_i avant B_{i+1}. $(1 \leqslant i \leqslant m-1)$.

En ce qui concerne le choix des clauses, Prolog choisira toujours les clauses dans l'ordre d'écriture du programme c.à.d. de haut en bas. Pour les clauses

(1) $B \leftarrow A_1 \wedge \ldots \wedge A_m$

(2) $B \leftarrow B_1 \wedge \ldots \wedge B_t$.

(n) $B \leftarrow C_1 \wedge C_2 \ldots \wedge Cp$.

le choix de la procédure de numéro i se fera toujours avant le choix de la clause i+1 ($1 \leqslant i \leqslant n-1$)

En cas d'échec (c.à.d. lorsque nous sommes en présence d'une SLD-dérivation avec échec), et ce, dans le but de prendre en compte tous les choix possibles, le dernier choix effectué est remis en cause. On revient donc à l'état qui précédait ce dernier choix c.à.d. que nous reprenons la question telle qu'elle était avant ce choix et nous effectuons un nouveau choix de clauses en excluant celles qui ont déjà fait l'objet d'un choix. Nous noterons que ce mécanisme est également mis en oeuvre pour la recherche de toutes les solutions à une question.

Le programmeur dispose donc pour exprimer le contrôle de l'ordonnancement des choix ainsi que de l'ordonnancement des littéraux dans la partie droite d'une clause. Il dispose également d'un certain nombre de prédicats évaluables. Les prédicats évaluables sont des prédicats primitifs du langage de programmation. Nous ne présenterons ici que le "Cut" (noté "!") qui est un mécanisme de contrôle très puissant bien que très controversé.

Le "Cut" est le prédicat de contrôle par excellence. Son effet peut être caractérisé de la manière suivante. Considérons une clause Prolog $A :- A_1, A_2, \ldots, A_{i-1}, !, A_i, \ldots, A_n$. (1)

Le "Cut" est une formule atomique qui réussit toujours, c.à.d. que lorsque le "Cut" est le littéral choisi, la question a toujours un descendant qui est la question dont on a supprimé le "Cut". Nous savons qu'avant de considérer le "Cut", lorsque la clause (1) est choisie, nous avons résolu $A_1, A_2, \ldots, A_{i-1}$. Lorsque nous considérons le "Cut", tous les choix effectués depuis la sélection de la clause (1) jusqu'au "Cut" ne seront plus remis en cause au cours du backtracking. Nous ne considérerons plus les autres clauses qui pouvaient être choisies si la clause (1) conduisait à une SLD-dérivation avec échec, ni les autres possibilités (c.à.d. les autres choix qui donneraient d'autres SLD-réfutation) de $A_1, \ldots A_{i-1}$.

Si SLD-résolution était cohérent et complet, il n'en sera plus de même pour Prolog. En effet sa stratégie de recherche en profondeur d'abord peut l'amener sur des SLD-dérivations infinies. La figure 3 illustre ce problème.

De même l'utilisation du "Cut" peut amener une autre forme d'incomplétude.

En effet, le "Cut" permet de supprimer l'exploration de sous-arbres entiers de l'espace de recherche. Une utilisation maladroite du "Cut" peut nous empêcher de trouver une solution ; dans ce cas, Prolog peut répondre "non" alors qu'il existait une réponse.

Enfin, nous devons attirer l'attention sur le problème de l'"occur-check" qui intervient lors de l'unification. Prolog ne vérifie pas si une variable est instantiée à un terme qui contient cette variable et ce pour des raisons d'efficacité.
Par exemple, Prolog permet à P(x,x) et P(y,f(y)) de s'unifier.
Cette absence d' "occur-check" a pour conséquence de supprimer la cohérence de Prolog et permet de trouver des réponses qui ne sont pas des conséquences logiques des clauses du programme.
Cependant si ces résultats peuvent paraitre assez négatifs, il ne faut pas perdre de vue que Prolog est avant tout un langage de programmation et qu'en tant que tel ces choix lui permettent d'être implémenté efficacement.
On peut évidemment en conclure que Prolog n'est pas un langage logique et qu'il est nécessaire d'en connaître l'interpréteur pour pouvoir programmer ou encore que la démonstration de correction d'un programme Prolog devra, à un moment ou à un autre, prendre en considération le fonctionnement de l'interpréteur.

1.3.3 Sémantique déclarative des programmes logiques.

Une des caractéristiques de la programmation en logique est que les programmes admettent une sémantique déclarative en plus de la sémantique procédurale. La sémantique déclarative d'un programme logique est donnée par la sémantique en logique du premier ordre des formules bien formées qui constituent le programme.

Un programme logique peut alors être utilisé pour calculer toute information qui est une conséquence logique du programme. A cette sémantique déclarative correspond une lecture déclarative du programme.

Considérons une clause de Horn de la forme

$$R(t_1, t_2, \ldots, t_n) \leftarrow A_1 \wedge A_2 \wedge \ldots \wedge A_m \quad m \geqslant o \qquad (1)$$

où A_i $(1 \leqslant i \leqslant m)$ et $R(t_1, t_2, \ldots, t_n)$ sont des formules atomiques.

$R(t_1, t_2, \ldots, t_n)$ est appelé le conséquent de la clause.

Si $x_1, x_2, \ldots, x_k$ sont des variables de la clause (1), cette clause peut être lue de la manière suivante :

Pour tout $x_1, x_2, \ldots, x_n$, $R(t_1, t_2, \ldots, t_m)$ est vrai si A_1 et .... et A_m sont vrais. Nous pouvons également la lire, dans le

Soit le programme

P(a,b)

P(c,b)

P(x,z) ← P(x,y), P(y,z)

P(x,y) ← P(y,x) .

Considérons la question ← P(a,c)

L'interpréteur Prolog ne trouvera jamais la solution

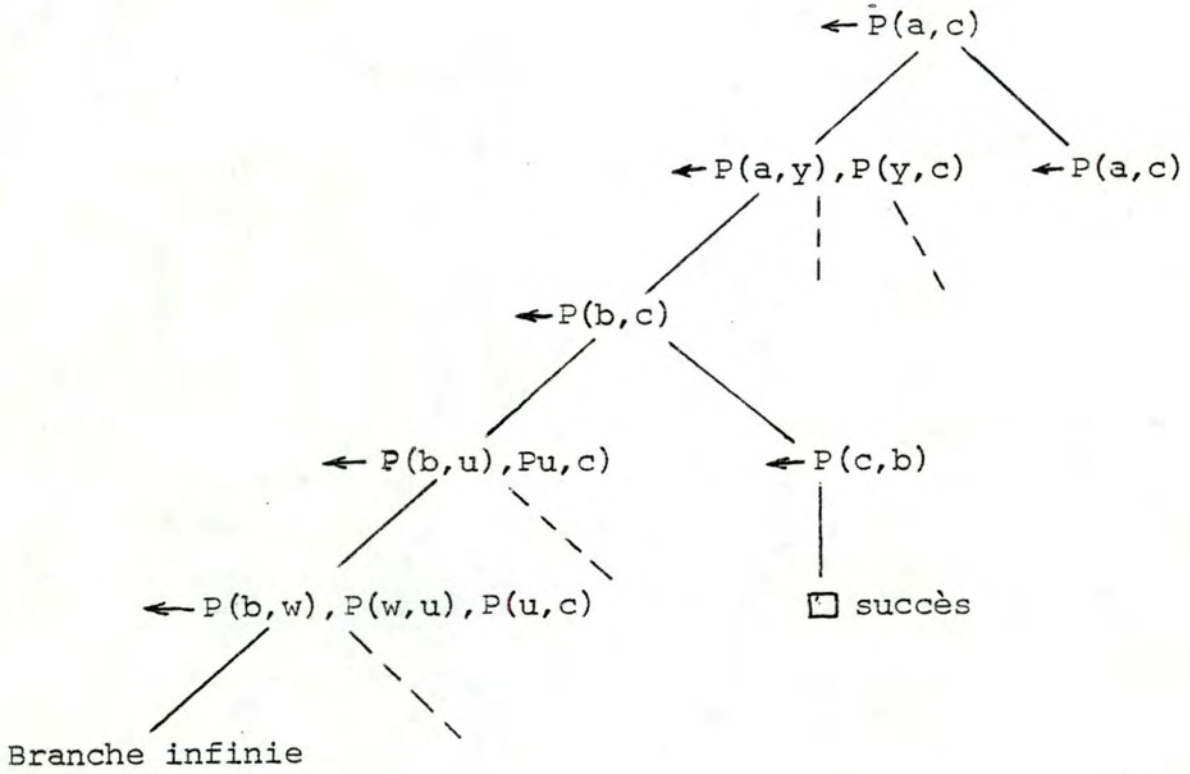


Figure 3

cas où $y_1, y_2, \ldots, y_i, \ldots$ sont des variables qui apparaissent uniquement dans l'antécédent $A_1 \wedge \ldots \wedge A_m$ et si $z_1, \ldots, z_i$ sont toutes les autres variables, de la manière suivante

Pour tout $z_1, z_2, \ldots, z_j$ $R(t_1, t_2, \ldots t_n)$ est vrai s'il existe $y_1, y_2, \ldots y_i$ telle que A_1 et A_2 et ... et A_m sont vrais.

exemple 1.

Convenons de la représentation suivante pour les listes

1) La liste vide est représentée par "nil".
2) Si a est un terme et x une liste, "cons(a,x) représente la liste dont le premier élément est a et les éléments restants sont représentés par x.

Considérons maintenant un programme de concaténation de listes. Le programme que nous appellerons P_1 peut être défini par

1) append(nil,x,x)
2) append(cons(u,x),y,cons(u,z)) ← append(x,y,z).

L'assertion (1) établit que la concaténation de la liste vide avec la liste x est la liste x quel que soit x.
L'implication (2) établit que pour tout x,y,z,u, si z est la concaténation de la liste x à la liste y, la liste cons(u,z) est la concaténation de la liste cons(u,x) et de la liste y.

Dans le cadre de la démonstration automatique de théorèmes, nous nous intéressons aux conséquences logiques d'un ensemble d'axiomes. Dans celui de la programmation en logique ce qui nous intéresse est d'avoir des réponses aux questions que nous posons. Pour ce faire, définissons les notions suivantes :

définition : soit P un programme logique
soit q une question
une substitution-réponse pour $P \cup \{Q\}$ est une substitution pour les variables de q.

définition : soit P un programme logique et θ une substitution
soit Q une question de la forme $\leftarrow A_1 \wedge A_2 \wedge \ldots \wedge A_k$
soit $x_1, x_2, \ldots, x_k$ les variables de $(A_1 \wedge A_2 \wedge \ldots \wedge A_k)\theta$
Nous dirons que θ est une substitution-réponse correcte pour $P \cup \{Q\}$

si $\forall x_1 \forall x_2 \forall x_3 \ldots \forall x_k ((A_1 \wedge \ldots \wedge A_k)\theta)$ est une conséquence logique de P.

Un système de programmation en logique peut alors être vu comme une boite noire qui recevant comme donnée un programme logique et une question renvoie soit une substitution-réponse correcte pour $P \cup \{Q\}$ soit "non" lorsque $P \cup \{Q\}$ est satisfaisable.

exemple 2.

Soit P_1 le programme logique vu précédemment.
Soit Q la question ← append(cons(a,cons(b,nil)),cons(c,nil),x)
x) où a,b,c sont des constantes et x une variable.
Cette question peut être interprétée comme une demande de calcul d'une sustitution.
réponse correcte pour $P_1 \cup \{Q\}$. Le résultat est la substitution
$\{cons(a,cons(b,cons(c,nil))) / x\}$
c.à.d. la concaténation de "cons(a,cons(b,nil))" et "cons(a,nil)".

Nous avons dit qu'un programme logique pouvait être utilisé pour calculer toute information qui est une conséquence logique du programme. Cela signifie qu'un même programme peut être utilisé pour différentes formes de questions admettant des combinaisons différentes pour les arguments de la formule atomique à gauche de la flêche d'implication dans une clause. Cette caractéristique de la programmation en logique est connue sous le nom de réversibilité.

exemple 3.

Considérons le programme P_1 et la question Q_1
← append(x,y,cons(a,cons(b,cons(c,nil)))).
ou a,b,c sont des constantes et x,y des variables.
Cette question est elle aussi interprétée comme une demande de calcul d'une substitution-réponse correcte pour P_1
Le résultat est l'une des substitutions
$\{(nil / x),(cons(a,cons(b,cons(c,nil))) / y)\}$
$\{(cons(a,nil) / x),(cons(b,cons(c,nil)) / y)\}$
$\{(cons(a,cons(b,nil)) / x),(cons(c,nil)) / y)\}$
$\{(cons(a,cons(b,cons(c,nil))) / x),(nil / y)\}$
telle que la concaténation de la substitution pour x et de la substitution pour y est la liste "cons(a,cons(b,cons(c,nil)))".

Si un programme possède une seule lecture déclarative, il possèdera autant de lectures procédurales qu'il existe d'utilisations possibles de ce programme.

exemple 4.

Le programme P_1 donne lieu à la lecture procédurale suivante.

Pour concaténer la liste x à la liste y

Si la liste x est la liste vide, il suffit de renvoyer y comme résultat

Sinon si la liste x est la liste "cons(u,x_1)" , il faut concaténer x_1 à y ce qui donne z et renvoyer cons(u,z) comme résultat.

Tout autre utilisation du programme logique demandera une autre lecture procédurale.

Nous aimerions, nous trouvant dans la présentation de la sémantique déclarative, faire une remarque concernant les fonctions en programmation logique. Les fonctions, ou plus généralement les termes, donnent aux langages de programmation logique de nombreuses caractéristiques des langages fonctionnels de manipulation symbolique comme Lisp. Cependant, il existe une différence fondamentale entre les fonctions Lisp et les fonctions en programmation logique. Les fonctions, en programmation logique, ne sont jamais évaluées. La fonction "cons(a,x)" signifie "la liste dont le premier élément est "a" et dont les éléments restants sont représentés par x". La fonction "cons" en Lisp a une tout autre signification. Elle est un constructeur de S-expressions qui recevant deux S-expressions renvoie une autre S-expression qui est la paire pointée $(\alpha.\beta)$ où α, β sont les valeurs respectives des deux S-expressions.

Les fonctions en programmation logique doivent être vues comme des structures de données et l'équivalent des constructeurs et des sélecteurs en Lisp est réalisé par l'unification.

L'avantage de la sémantique déclarative est de permettre une abstraction du contrôle puisque la logique du premier ordre ne se préoccupe aucunement du contrôle. C'est ainsi qu'elle est à la base d'une méthodologie de programmation en logique proposée par Kowalski [KOWA 79] et que nous allons étudier ici.

1.3.4 Algorithme = Logique + Contrôle.

Il nous parait primordial de présenter la philosophie d'un outil

en même temps que l'outil lui-même pour bien percevoir ses fonctionnalités et pourquoi il a été conçu. C'est pourquoi nous tenterons de présenter brièvement les grandes étapes de construction d'un programme logique. Il n'existe bien évidemment aucune recette-miracle pour la construction d'un programme logique correct mais nous pouvons présenter une démarche qui nous semble adéquate.

Partant d'une spécification qui peut être de la forme définie dans [DEVI 85], la conception d'un programme logique donne lieu à deux étapes :

(1) la définition de la logique du programme c.à d. la connaissance nécessaire pour résoudre le problème.

(2) la définition du contrôle du programme c.à d. les informations nécessaires pour utiliser cette connaissance.

Un programme logique devrait d'abord être construit en définissant toute la logique du programme avant de passer à la spécification du contrôle.

La définition de la partie logique, appelée composant logique, consiste à spécifier une relation entre les arguments du conséquent d'une implication au moyen d'une ou plusieurs formules atomiques qui forment les antécédents. Les antécédents forment un ensemble c.à.d. que l'ordre dans lequel ils sont formulés n'a aucune importance. Le résultat de cette étape est seulement un ensemble de définitions qui sont nécessaires pour résoudre le problème. C'est ainsi qu'un programme efface (T_1, T_2, T_3) qui exprime la relation suivante

"T_3 est la liste T_2 dont on a supprimé la première occurence de T_1" aura comme composant logique (au programme logique)

efface (T_1, T_2, T_3) ←
T_2 = cons (T_1, T_4) &
T_3 = T_4.

efface (T_1, T_2, T_3) ←
T_2 = cons (T_4, T_5) &
$T_4 \neq T_1$ &
efface (T_1, T_5, T_6) &
T_3 = cons (T_4, T_6).

Dans cette définition, l'ordre des clauses n'a aucune importance.
De même, le corps de chaque clause est une conjonction et l'ordre des formules atomiques n'a donc aucune importance.
Lorsque le problème à résoudre est plus conséquent, l'obtention du composant logique peut se faire par raffinements successifs [WIRT 71].

Le problème est alors résolu en faisant apparaître des sous-problèmes, en spécifiant ces sous-problèmes et en les utilisant pour résoudre le problème initial. Le processus recommence alors avec les sous-problèmes jusqu'à ce que tous les sous-problèmes soient résolus uniquement en termes des prédicats primitifs.

La définition du contrôle, que nous appellerons composant de contrôle, consiste à exprimer les informations de contrôle dans le but d'obtenir un programme efficace et réellement utilisable. Certains prétendent que la définition du contrôle a pour but d'obtenir un programme. Pour eux, le composant logique n'est pas un programme et la définition du contrôle consiste à faire du composant logique un programme. Nous préférons considérer que le composant logique est un programme, inefficace certes, mais un programme puisqu'il nous donne le moyen de calculer la relation.

Le composant de contrôle va nous indiquer comment exploiter le composant logique pour en faire un programme du langage de programmation considéré. Ces informations de contrôle constituent une connaissance sur la manière dont on peut utiliser efficacement le composant logique. Comme le composant logique exprime lui-même une connaissance à propos de la relation que l'on veut calculer, nous parlons parfois de méta-connaissance pour les informations de contrôle. Le problème du contrôle est un problème qui reste ouvert en programmation logique. Plusieurs formes de contrôle ont été élaborées. Le contrôle pragmatique consiste à connaître la manière de travailler de l'interpréteur et à "s'arranger" de telle sorte que celui-ci fasse bien ce que l'on lui demande. L'ordonnancement des clauses du programme sont des exemples de contrôle pragmatique. La deuxième forme de contrôle consiste à incorporer des informations de contrôle dans le programme. Le "Cut" de Prolog est un exemple de ce type de contrôle. De même, le contrôle par annotations constitue un représentant non négligeable de cette catégorie. Une troisième forme de contrôle consiste à séparer le contrôle du programme logique et à l'exprimer dans un langage séparé. Cette forme de contrôle est appelée contrôle par méta-règles, exprimant par là que le contrôle est énoncé dans un langage distinct du programme. D'autres formes de contrôle existent. Disposant d'un langage de programmation logique, il est assez aisé de réécrire un interpréteur de ce langage qui sera adapté à notre problème. Chaque forme de contrôle a ses avantages : le contrôle pragmatique, sa facilité d'implémentation et son efficacité, le contrôle par méta-règles, ses possibilités d'expressions. Le contrôle par annotations permet d'augmenter le contrôle pragmatique tout en conservant une grande

efficacité. Il n'y a pas encore de réponses définitives à ce problème et il n'y en aura peut-être jamais, la réponse dépendant parfois du problème à résoudre. Dans ce qui suit, nous examinerons brièvement le contrôle de Prolog, le contrôle par annotations et le contrôle par méta-règles.

Les informations de contrôle de Prolog.

La stratégie de Prolog a été présentée ci-dessus. Les moyens d'expression du contrôle sont principalement les suivants.

- l'utilisateur peut ordonner les clauses pour définir un ordre de préférence entre ces clauses. Toute clause qui précède une autre clause dans l'ordre d'écriture du programme sera essayée avant cette dernière.
- l'utilisateur peut ordonner les formules atomiques à l'intérieur d'une partie droite de clause et choisir ainsi la combinaison qui parait la plus efficace eu égard à l'exécution. Cette possibilité permet d'écrire des programmes prolog qui n'ont pas de programmes logiques ou composant logique correspondant c.à.d. où l'ordre des formules atomiques (il sera plus adéquat de parler d'appels de procédure) a une signification sémantique et où tout autre ordre définirait une autre relation entre les arguments.
- enfin, le prédicat évaluable "Cut" permet à l'utilisateur de supprimer des parties entières de l'arbre de recherche qui devraient ne rien ajouter à la solution du problème.

L'expérience a montré que ces moyens d'expressions du contrôle étaient suffisants même s'ils ont fait l'objet d'un certain nombre de critiques, spécialement le "Cut". L'avantage de ces mécanismes est leur simplicité de mise en oeuvre.

Le programme logique précédent, si l'usage que l'on veut en faire est de générer une liste T_3 à partir d'un terme T_1 et d'une liste T_2, donne lieu au programme Prolog suivant

```
efface(B1,[B1/B2],B2):- ! .
efface(B1,[B2/B3],[B2 B5]):- efface(B1,B3,B5) .
```

Le "Cut" exprime que, lorsque nous sommes arrivés là, il n'est plus nécessaire de chercher d'autres solutions car il n'y en aura plus. Son absence conduirait, en cas de remise en cause du choix, à générer des

solutions qui ne seraient pas des solutions définies par le programme logique. Lorsque nous choisissons la deuxième clause, cela signifie que la première n'a pas été choisie et donc que $B_1 \neq B_2$. De même l'appel récursif efface (B_1, B_3, B_5) considèrera évidemment d'abord la première clause. S'il n'en était pas ainsi (ie : les clauses étaient ordonnées de manière différentes) le programme Prolog ne correspondrait plus au programme logique.

Le_contrôle par annotations.

Ce type de contrôle utilise des annotations à l'intérieur des clauses pour exprimer le contrôle. Le premier type d'utilisation de ces annotations consiste à utiliser de manière efficace les antécédents d'une clause. Considérons la clause Prolog

G(x,y) :- P(x,z), P(z,y) où P(x,z), P(z,y) font référence à des assertions en assez grand nombre. Cette clause est efficace lorsque nous l'utilisons pour déterminer k tel que G(a,k). Par contre, elle l'est beaucoup moins lorsque nous voulons l'utiliser pour déterminer k tel que G(k,a) car elle va commencer par générer tous les couples (x,y) tel que P(x,y) est vrai.
C'est pourquoi, dans ce cas, nous pouvons remplacer la procédure Prolog par la procédure

```
[G(x? y) :- P(x,z),P(z,x).
 G(x,y?) :- P(z,y),P(x,z).].
```

L'annotation '?' exprime la condition que x soit totalement instantiée (i.e : ne contienne plus de variables).

Les annotations permettent également de mettre en oeuvre un mécanisme de co-routines entre les différentes formules atomiques d'une partie droite de clause. Ce mécanisme est souvent indispensable lorsque l'on désire comparer deux ensembles ou lorsque l'on désire générer une solution composée d'un ensemble d'éléments qui doivent vérifier des propriétés entre eux. Dans le premier cas, il nous évite de générer tout le premier ensemble pour ensuite vérifier s'il est compatible avec le second. Dans le second, il nous évite de générer une proposition de solutions qui s'avérera fausse. Dans le premier cas, dès que l'on va générer un élément, on va regarder s'il existe un élément identique dans l'autre ensemble. Dans le second cas, quand on ajoute un élément pour former une solution, on s'assure qu'il vérifie les propriétés par rapport aux éléments déjà introduits. Le lecteur intéressé pourra se reporter à [CLAR 79] , [CLAR 80] .

Les annotations sont des moyens d'expression du contrôle qui complètent une stratégie du contrôle pragmatique.

Le_contrôle_par_méta-règles.

Certaines stratégies permettent d'exprimer le contrôle indépendamment de la définition de la logique du programme. L'expression du contrôle se fait dans un certain langage qui peut d'ailleurs très bien être la logique ou les clauses de Horn. Ces stratégies ont pour conséquence que la séparation de la logique et du contrôle n'est plus seulement une séparation conceptuelle mais se reflète également dans l'expression du programme final. Le programme qui en résulte est alors plus lisible et plus aisément modifiable. Ces stratégies offrent généralement des moyens d'expression du contrôle plus développés que les stratégies précédentes. L'utilisateur sera plus à même d'exprimer entièrement toutes ces connaissances sur la manière dont il faut utiliser le composant logique. Nous ne pourrons cependant pas "compiler" le programme logique et le composant de contrôle pour générer par exemple un programme dans une des deux stratégies définies précédemment. La raison en est que le contrôle porte ici sur tout l'ensemble de l'arbre de recherche et ne se limite pas au choix des clauses ou des formules atomiques dans les clauses. Le lecteur intéressé peut se reporter à [DINC 80] , [GALL 82] , [DAVI 80a] , [DAVI 80] .

1.3.5 Critiques et mises en garde.

Nous aimerions terminer par un certain nombre de réflexions qui nous ont semblé importantes et qui concernent la programmation en logique. Ces réflexions sont inspirées à la fois de l'étude théorique qui précède et de l'utilisation d'un langage de programmation en logique.

1. Un_programme_logique_n'est_pas_une_spécification.

Le programme logique ne peut, en toute généralité, être considéré comme une spécification adéquate pour être utilisée par un programmeur [DEVI 85] . Tout d'abord un certain nombre d'informations sont absentes de ce programme final. Nous pensons particulièrement à l'usage que l'on désirera faire du programme. Ensuite, même si cet aspect est mis de côté, nous signalerons qu'un programme logique dès qu'il atteint une certaine taille devient tout aussi illisible que tout autre programme. Enfin, pour un certain nombre de programmes, indispensables dès que l'on veut utiliser le langage pour un problème concret, il est difficile et parfois même impossible de trouver un programme logique qui résoud le problème. Néanmoins, le passage par un tel programme aide fortement à la construction

d'un programme correct. Le lecteur pourra se reporter aux programmes réalisés dans le cadre de ce mémoire. Ils utilisent la méthode de spécification décrite dans [DEVI 85] . Ensuite pour chaque programme, nous donnons le programme logique pour passer, enfin, au programme prolog. Lorsqu'aucun programme logique n'est mentionné, cela signifie que nous avons considéré que cela avait peu ou pas de sens de parler de programme logique pour ce type de programme.

2. Les informations de contrôle sont et seront toujours nécessaires.

Les réflexions que nous avons émises au sujet de la séparation de la logique et du contrôle pourraient susciter des réflexions comme "les informations de contrôle sont-elles vraiment nécessaires ?". "N'est ce pas un pur problème d'optimisation ?". "Ne pourrait-on pas imaginer de construire, dans les années qui viennent avec le développement et l'abaissement des coûts du hardware, un interpréteur qui pourrait exécuter le programme de manière efficace sans devoir exprimer ces informations ?". Ces réflexions sont guidées par la constatation que le programme logique et le programme final ne sont pas vraiment très différents et l'intuition que, moyennant un peu de temps CPU, l'interpréteur sera à même de les retouver. C'est notre opinion que de dire que le fossé entre le programme logique et le programme final ne pourra être comblé efficacement par l'interpréteur, que ces informations font partie intégrante du programme et qu'elles ne doivent pas être considérées comme un simple problème d'optimisation. Ces informations sont et seront toujours nécessaires si nous voulons construire un programme qui soit réellement efficace et utilisable. Cependant, les langages de programmation peuvent prendre en compte un grand nombre de ces informations les moyens d'expression du contrôle peuvent prendre une forme qui soit plus orientée vers le programmeur et moins orientée vers la machine mais en dernier ressort ces informations resteront toujours sous la responsabilité du programmeur. Nous pensons, en effet, que quel que soit le degré de sophistication de l'interpréteur, il sera toujours plus facile au programmeur de trouver les informations de contrôle pertinentes pour son problème que pour l'interpréteur de les découvrir.

3. Sur l'équivalence entre les sémantiques déclarative et opérationnelle.

L'équivalence entre la sémantique déclarative et la sémantique opérationnelle ainsi que la séparation entre la logique et le contrôle

sont parmi les qualités les plus soulignées de la programmation en logique. Cependant, nous allons voir que l'une et l'autre sont soumises à certaines restrictions :

On remarquera dans un premier temps que la présence de prédicats évaluables (les primitives du langage) ruine les résultats de complétude et de cohérence entre les deux sémantiques. Il faudrait en effet, que les prédicats évaluables soient définis eux-mêmes sous forme de clauses si l'on veut que ces résultats soient valables. Si cette remarque parait de peu d'importance (on peut par exemple, considérer que ces prédicats évaluables sont définis sous forme de clauses), il met en évidence la question suivante :

"Tous les prédicats évaluables sont-ils exprimables sous forme de clauses de Horn ?".

Cette question doit être reliée au théorème de Gödel qui énonce qu'il n'existe pas de formalisation complète et cohérente de l'arithmétique. Autrement dit, si on définit un ensemble d'axiomes qui admet l'arithmétique comme modèle, cet ensemble admettra d'autres modèles de sorte qu'il sera possible de trouver une formule vraie pour l'arithmétique qui ne sera pas vraie dans d'autres modèles. Par conséquent, cette formule ne sera pas vraie dans tous les modèles, sa négation non plus, et on dira qu'elle est indécidable.

Pratiquement, le théorème de Gödel nous indique que l'on pourra construire un programme logique qui calcule une et une seule relation entre les arguments du conséquent des clauses qui le constituent et qui ne fournira jamais de résultats. Supposons, par exemple, que nous disposions du programme "append" et que nous désirions construire un programme qui, pour un argument quelconque, renvoie toujours la liste vide. Le programme logique suivant répond à cette spécification

$F(x,res) \leftarrow F(x,y) \& F(x,z) \& append(y,z,res)$

Bien sûr, ce n'est pas là, la manière la plus simple de construire le programme logique mais ce programme est parfaitement correct, complet et cohérent du point de vue de la logique des prédicats du premier ordre. Cependant aucun interpréteur ne fournira le résultat attendu.

Ce résultat a une conséquence immédiate, à la fois sur l'équivalence

entre les deux sémantiques, déclarative et opérationnelle, et sur l'équation "algorithme = logique + contrôle". Si la première devrait être claire, la seconde demande une petite explication. En effet, ce résultat impose une condition sur la forme du composant logique. Le composant logique ne peut se contenter de donner une définition en logique des prédicats du premier ordre correspondant à la spécification ; cette définition devra, de plus, être un algorithme qui permette de calculer la relation définie entre les arguments du conséquent de la clause (ou des clauses). C'est ainsi que le composant logique de "append" vérifie cette condition tandis que la "clause" f(x,res)← f(x,y) & f(x,z) & append (y,z,res)" ne la vérifie pas. Le composant logique sera donc nécessairement un programme et, à ce titre, il nous renseigne sur la manière de procéder pour résoudre le problème. Par conséquent, le composant logique introduit, dès le départ, la notion de contrôle. De là à conclure que la séparation entre la logique et le contrôle est complètement fausse ou artificielle, il n' a qu'un pas.

Il nous semble cependant plus juste de dire que par rapport à la programmation conventionnelle, la programmation logique offre une plus grande liberté pour l'expression de la première version du programme (composant logique), nous évitant de prendre en considération tout le contrôle en une fois.

CHAPITRE 2. LE MODELE DE BASE DE DONNEES RELATIONNEL.

2.0 Introduction.

Le modèle relationnel a fait l'objet d'un grand nombre de recherches depuis son introduction par Cood [COOD 71] . Des systèmes relationnels sont aujourd'hui commercialisés tandis que le modèle relationnel jouit d'un tel succès que d'aucuns le considèrent comme le modèle qui retiendra le plus d'attentions dans les années à venir. Les raisons de ce succès doivent être trouvées dans les objectifs assignés au modèle relationnel qui sont principalement

- l'objectif d'indépendance de données qui consiste en une séparation claire et précise entre les niveaux logique et physique d'une base de données pour ce qui est de la définition de la base de données, de la recherche et de la manipulation des données.
- l'objectif de communication qui consiste à présenter un modèle simple qui puisse servir de représentation commune aux utilisateurs et programmeurs.
- l'objectif de manipulation d'ensemble de données qui consiste à introduire des langages d'interrogation manipulant des ensembles de données.
- l'objectif de définition théorique qui consiste à définir une base théorique pour le modèle.

Avant de présenter le modèle relationnel, il convient de définir ce que l'on entend par modèle de données. Selon Cood [COOD 82] , un modèle de données est constitué de trois composants.

(1) un composant structuration constitué d'une collection de types distincts de structures de données.

(2) un composant manipulation qui est constitué d'une collection d'opérateurs ou de règles d'inférence qui peuvent être appliquées à tout instance valide des types de données définis dans (1) dans le but de retrouver, de dériver ou de modifier les données de toute instance.

(3) un composant d'intégrité qui est constitué d'une collection de règles d'intégrité générales qui définissent soit explicitement soit implicitement l'ensemble des états cohérents de la base de données et/ou l'ensemble des changements d'état admissibles de la base de données. Ces règles sont générales dans le sens où elles s'appliquent à toutes bases de données utilisant ce modèle.

Le modèle relationnel est un modèle de données au sens de la définition présentée ci-dessous. Le composant structuration est constitué de domaines, de relations, d'attributs et ainsi de suite. Le composant manipulation est constitué de l'algèbre relationnelle qui transforme des relations en d'autres relations. Le composant intégrité est constitué de deux règles : l'intégrité des identifiants et l'intégrité référentielle. Cependant, dans toute application spécifique, il peut être nécessaire de définir de nouvelles contraintes et par là, limiter le nombre d'états valides de la base de données et/ou restreindre les transitions entre états valides.
Dans ce qui suit, nous présenterons successivement chacun des composants du modèle relationnel.

2.1 Le composant structuration.

Le concept de base du modèle relationnel est la notion de relation. Elle peut être définie de la manière suivante.

Définition : un domaine est un ensemble de valeurs. L'ensemble des entiers ou l'ensemble des chaînes de caractères de longueur 20 sont des domaines. Nous verrons ultérieurement que l'on peut assurer des notions plus sémantiques aux domaines. Les domaines sont alors des employés, des départements, etc

Définition : le produit cartésien des domaines $D_1, D_2, \ldots, D_u$ noté $D_1 x D_2 x \ldots x D_u$ est l'ensemble de tous les **U-tuples** $(U_1, U_2, \ldots, U_u)$ tel que $U_i \in D_i (1 \leqslant i \leqslant u)$

Définition : Une relation est un sous-ensemble du produit cartésien d'un ou de plusieurs domaines.

Les membres d'une relation sont appelés les tuples. Chaque relation qui est un sous-ensemble du produit cartésien $D_1 x D_2 x \ldots x D_k$ est dit d'arité k. Un tuple $(U_1, U_2, \ldots, U_k)$ a k composants, le ième $(1 \leqslant i \leqslant k)$ étant U_i.
Nous remarquerons que telles que nous les avons définies, les relations vérifient les propriéteés suivantes

1° tous les tuples sont distincts et l'ordre des tuples dans une relation n'est pas défini puisque nous sommes en présence d'un ensemble.

2° l'ordre des composants dans un tuple est significatif car

deux composants peuvent prendre leur valeur sur un même domaine.

Il est possible de supprimer l'ordre des composants en définissant la notion d'attribut. L'attribut est le rôle joué par un domaine dans une relation. Dans une relation, le rôle d'un domaine particulier est désigné par un nom appelé "nom d'attribut".
Nous pouvons maintenant définir deux nouvelles notions.

Définition : On appelle schéma d'une relation de nom 'R' dont les noms d'attribut sont $A_1 A_2 \ldots A_k$,

$$R(A_1:D_1,\ A_2:D_2,\ \ldots\ ,A_k:D_k)$$

où D_i $(1 \leqslant i \leqslant k)$ est le domaine de l'attribut de nom A_i.

Définition : On appelle extension d'une relation donnée l'ensemble des tuples qui appartiennent à la relation à un instant donné.

Une relation peut être visualisée sous forme d'un tableau. Chaque ligne du tableau est un tuple et les colonnes représentent les composants. Chaque colonne reçoit un nom qui est le nom d'attribut.
Dans ce qui suit, nous supposerons que le nom d'attribut est identique au nom de domaine excepté lorsqu'il est explicité.
Un exemple de relation est donné à la figure 2.1.

Définition : Un identifiant d'une relation est une liste d'attributs telle qu'il n'y a pas dans cette relation deux tuples qui possèdent les mêmes valeurs pour les attributs respectifs de cette liste.

Nous remarquerons qu'une relation étant un ensemble elle possède toujours un identifiant constitué de la liste de tous les attributs. C'est pourquoi, nous définirons également la notion d'identifiant strict.

Définition : Un identifiant strict d'une relation est une liste d'attributs telle que cette liste soit un identifiant et telle qu'aucune de ces sous-listes ne soit un identifiant.

Dans le modèle de Cood, on ne parle pas d'identifiant mais de clé. Nous préférons parler d'identifiant que de clé étant donné que ce dernier terme à des connotations d'implémentation [HAIN 85] . C'est ainsi que Cood parle de clés candidates pour désigner les identifiants stricts. De plus, parmi ces clés candidates, une clé reçoit un status particulier

Soit la relation qui exprime la population d'une ville d'une certaine province.
Le schéma de la relation est défini par

pop(population,ville,province)

Son extension peut être définie à un instant donné comme

pop	population	ville	province
	100.000	Namur	Namur
	1.000.000	Bruxelles	Brabant

figure 2.1

et est appelée "clé primaire". Dans ce qui suit, nous parlerons d'identifiant primaire.
Remarquons enfin que les domaines tels que nous les avons définis sont simples. La relation est sous première forme normale. Cependant, on admet généralement qu'un composant d'un tuple puisse prendre une valeur spéciale appelée "valeur nulle". Cette valeur n'a pas une sémantique précise. Elle peut représenter des valeurs inconnues ou des valeurs non avenues.
De nombreuses recherches ont été effectuées en vue de savoir quelles formes devaient avoir les relations. On a ainsi abouti à la définition d'une série de formes normales. Le lecteur intéressé par la définition de ces formes peut se reporter à [KENT 83] [DATE 79] [ULLM 80]

2.2 Le composant manipulation.

Le composant manipulation du langage relationnel est donné par l'algèbre relationnelle avec pour but de remplir l'objectif de manipulation d'ensembles de données. L'algèbre relationnelle est constituée d'une collection d'opérations sur les relations. Chaque opération prend une ou plusieurs relations comme opérandes et produit une autre relation comme résultat. Comme le résultat d'une opération est une relation, cette relation peut à son tour, se voir appliquer des opérations.

L'objectif de l'algèbre relationnelle n'est pas d'être un langage standard auquel tous les modèles relationnels devraient souscrire mais plutôt de définir une capacité minimale de manipulation. Un système de gestion de base de données doit avoir un langage d'interrogation qui a au moins la puissance de l'algèbre relationnelle sans utiliser des instructions d'itération et la récursivité. Toutes les questions ne sont pas exprimables dans l'algèbre relationnelle (par ex : la fermeture transitive d'une relation) mais elle est destinée à être utilisée en conjonction avec un langage de programmation (ou plusieurs). Enfin, pour être en accord avec l'objectif d'indépendance des données, un langage de base de données relationnelle ne doit pas faire apparaître de notions physiques. C'est le S.G.B.D. qui prendra en charge la manière dont on accédera aux données.

Dans ce qui suit, nous présentons l'algèbre relationnelle. Nous distinguerons les opérations traditionnelles sur les ensembles (union, intersection), des opérations spécifiques au modèle relationnel (jointure, projection, sélection). Notons cependant, que seules cinq opérations (union, différence, produit cartésien, projection, sélection) sont

nécessaires. Les autres opérations n'ajoutent rien à la puissance d'expression du langage mais permettent de faciliter l'expression des questions.
Dans ce qui suit, nous ne supposerons pas que les colonnes sont nommées par un nom d'attribut. L'ordre des composants est donc significatif.
La conversion des opérations au cas où les colonnes ont un nom d'attribut est relativement aisée.

2.2.1 Les opérations sur ensemble.

Ces opérations sont : l'union, l'intersection, la différence et le produit cartésien. Chacune d'entre elles, excepté le produit cartésien, nécessite que les relations qui servent d'opérandes soient union-compatibles c-à-d que les opérandes soient de même arité k et que les composants j ($1 \leqslant j \leqslant k$) des deux relations appartiennent au même domaine.
Pour chaque opération, des exemples sont présentés en figure 2.2

2.2.1.1 L'union.

L'union des relations R et S noté $R \cup S$ est l'ensemble des tuples qui sont dans R ou dans S ou encore dans R et S.

2.2.1.2 La différence.

La différence de deux relations R et S, noté $R \setminus S$, est l'ensemble des tuples qui appartiennent à R et pas à S.

2.2.1.3 L'intersection.

L'intersection de deux relations R et S, noté $R \cap S$, est l'ensemble des tuples qui appartiennent à la fois à R et à S.

$R \cap S = R \setminus (R \setminus S)$.

2.2.1.4 Le produit cartésien.

Considérons deux relations R_1 et R_2 d'arité respective k_1 et k_2. Le produit cartésien de R et S, noté RxS est l'ensemble des (k_1+k_2)-tuples dont les k_1 premiers composants forment un tuple dans R et dont les k_2 derniers composants forment un tuple dans S. RxS est donc une relation d'arité k_1+k_2.
On remarquera cependant que cette définition ne correspond pas à la notion mathématique de produit cartésien.

R

a	b	c
d	a	f
c	b	d

S

b	g	a
d	a	f

R ∪ S

a	b	c
d	a	f
c	b	d
b	g	a

R \ S

a	b	c
c	b	d

R ∩ S

d	a	f

R × S

a	b	c	b	g	d
a	b	c	d	a	f
d	a	f	b	g	d
d	a	f	d	a	f
c	b	d	b	g	a
c	b	d	d	a	f

$\Pi_{1,3}(R)$

a	c
d	f
c	d

$\sigma_{2='b'}(R)$

a	b	c
c	b	d

figure 2.2.

2.2.2 Les opérateurs spécifiques.

2.2.2.1 La projection.

La projection prend comme opérande une relation, retient un certain nombre de composants de cette relation et les ordonne d'une certaine manière.

Plus précisément, si R est une relation d'arité k, $\Pi_{i_1,i_2,\ldots,i_m}(R)$ où $1 \leqslant i_j \leqslant k$ $(1 \leqslant j \leqslant m)$ et où $i_j \neq i_k$ si $j \neq k$, est la projection de R sur les composants $i_1,i_2,\ldots,i_m$.

$\Pi_{i_1,i_2,\ldots,i_m}(R)$ représente l'ensemble des m-tuples $a_1,a_2,\ldots,a_m$ tel qu'il existe un k-tuple $b_1,b_2,\ldots,b_k$ dans R tel que $a_j = b_{i_j}$ $(1 \leqslant j \leqslant m)$.

Un exemple de projection peut être trouvé à la figure 2.2.

2.2.2.2 La sélection.

La sélection renvoie un sous-ensemble de tuples d'une relation donnée qui vérifient un prédicat donné. Le prédicat est constitué d'une combinaison de termes, chaque terme étant une simple comparaison qui peut être évaluée à vrai ou à faux en ne considérant que ce tuple. Si un terme comprend une comparaison entre les valeurs de deux composants du tuple alors ces composants doivent appartenir au même domaine. Si F est un prédicat, si R est une relation, la relation est notée $\sigma_F(R)$.

Un exemple de sélection peut être trouvé à la figure 2.2.

2.2.2.3 La jointure.

Les θ-jointures de R et de S sur les colonnes i et j respectivement est l'ensemble des tuples du produit cartésien de R et S tels que la comparaison A θ B est vérifiée avec

A le ième composant de la relation R
B le jème composant de la relation S
un opérateur de comparaison pris dans $\{ <, \leq, >, \geqslant, =, \neq \}$

Par conséquent, la θ-jointure de deux relations R et S sur les colonnes i et j respectivement, notée $R \underset{i\,\theta\,j}{\bowtie} S$ où θ est un opérateur de comparaison, est une abréviation de

$\sigma_{i\,\theta\,(r+j)}(R \times S)$ où r est l'arité de R.

Si θ désigne l'égalité, l'opération est appelée équi-jointure.

La jointure est illustrée à la figure 2.3.

Les θ-jointures ne sont pas d'une grande utilité pratique. Cependant,

R			
	1	2	3
	4	5	6
	7	8	9

S		
	3	1
	6	2

R ⋈ S 2 < 1					
	1	2	3	3	1
	1	2	3	6	2
	4	5	6	6	2

figure 2.3.

R	A	B	C
	a	b	c
	d	b	c
	b	b	f
	c	a	d

S	B	C	D
	b	c	d
	b	c	e
	a	d	d

R ⋈ S	A	B	C	D
	a	b	c	d
	a	b	c	e
	d	b	c	d
	d	b	c	e
	c	a	d	b

figure 2.4.

il existe une forme de jointure tellement commune qu'il est adéquat d'en avoir une abréviation. Il s'agit de la jointure naturelle .

2.2.2.4 La jointure naturelle.

Cette opération est définie, pour deux relations R et S, lorsque R et S ont les colonnes nommées par des attributs. La jointure naturelle des deux relations, notée $R \bowtie S$, est définie par

(1) calculer le produit cartésien RxS.
(2) pour chaque nom d'attribut A qui désigne à la fois une colonne dans R et une colonne dans S, ne retenir que les tuples de RxS tels que leurs valeurs soient identiques dans les colonnes R.A ou S.A où R.A désigne le nom de la colonne de RxS correspondant à la colonne A de R et S.A désigne le nom de la colonne de RxS correspondant à la colonne B de S.
(3) pour tout pareil attribut A, supprimer la colonne S.A.

La définition de la jointure naturelle en algèbre relationnelle, si $A_1, A_2, \ldots, A_k$ sont les attributs utilisés dans R et dans S à la fois, est donnée par

$$\Pi_{i_1,\ldots,i_m} \sigma_{R.A_1=S.A_1 \quad \ldots \quad R.A_k=S.A_k} (RxS)$$

où $i_1, \ldots, i_m$ est la liste des composants de RxS dans l'ordre du produit cartésien auquel on a retiré $S.A_1 \ldots S.A_k$. La jointure naturelle est illustrée à la figure 2.4.

2.2.2.5 La division.

Cette opération divise une relation dividende A d'arité m+n par une relation diviseur B d'arité n et produit comme résultat une relation d'arité m. Le (m + i)ème composant de A et le ième composant de B ($1 \leqslant i \leqslant m$) doivent être définis sur le même domaine. Considérons les m premiers composants de A comme un composant unique noté x et les n derniers composants comme un composant unique noté y. Dès lors, A peut être vue comme un ensemble de paires de valeurs $\langle x,y \rangle$. De la même manière, B peut être vue comme un ensemble de valeurs $\langle y \rangle$.
Le résultat de la division de A par B, noté $A \div B$, est l'ensemble des valeurs x telle que la paire $\langle x,y \rangle$ apparaît dans A pour toutes les valeurs y apparaissant dans B.
La division de R par S peut s'exprimer en algèbre relationnelle par

$$R \div S = \Pi_{1,\ldots,m}(R) \setminus \Pi_{1,\ldots,m}((\Pi_{1,\ldots,m}(R) \times S) \setminus R).$$

La division est illustrée à la figure 2.5.

R				
	a	b	c	d
	a	b	e	f
	e	d	c	d
	e	d	e	f
	b	c	e	f
	a	b	d	e

S		
	c	d
	e	f

R ÷ S		
	a	b
	e	d

figure 2.5.

2.3 Le composant intégrité.

Le composant intégrité a pour objectif de s'assurer que la base de données soit une représentation aussi fidèle que possible du monde réel. Il est bien entendu impossible de s'assurer une représentation complètement fidèle. Si un utilisateur insère dans la base de données un fait exprimant que l'âge du client 'Dupont' est 32 alors que 'Dupont' a en réalité 34 ans, rien ne nous permettra à priori, de détecter cette incohérence. Cependant, il est possible de s'assurer un certain contrôle sur les informations mémorisées. Pour ce faire, on utilise des contraintes d'intégrité qui sont des propriétés que doivent vérifier les états ou les transitions entre états de la base de données. Les états de la base de données sont constitués par l'extension des relations et les contraintes d'intégrité définissant les états valides de la base de données ainsi que les transitions autorisées entre ces états valides.

De nombreuses classifications des contraintes d'intégrité ont été présentées dans la littérature. On distingue les contraintes sur un tuple ou un ensemble de tuples, les contraintes intra ou inter relations, les contraintes d'état et de transition ...
Remarquons que les dépendances fonctionnelles et multivaluées sont, elles-mêmes, des contraintes d'intégrité.

Une classification qui nous sera utile par la suite est la distinction entre contraintes d'état et contraintes de transition. Une contrainte d'état est une contrainte qui doit être vérifiée à tout moment par tout état de la base de données. La contrainte "L'age de toute personne est inférieur à 150" est une contrainte d'état. Une contrainte de transition est une contrainte qui doit être vérifiée lors du passage d'un état à l'autre de la base de données. Par exemple, "L'âge d'un employé ne peut que croître".
Il ne faut cependant pas croire que les seules contraintes de transition sont celles qui se réfèrent à des valeurs de tuples provenant de deux états distincts de la base de données. Le fait de se référer à deux états distincts est une condition qui identifie les contraintes de transition mais toutes les contraintes de transition ne la vérifient pas. Par exemple, la contrainte "On ne peut supprimer un client que si sa balance est nulle" ne se réfère aux valeurs que d'un seul état mais elle constitue une contrainte de transition.

Jusqu'au chapitre 4 de la deuxième partie, lorsque nous parlerons de contraintes d'intégrité, nous désignerons les contraintes d'état. Ce chapitre étudiera un peu plus en détail les contraintes de transition.

D'autre part, il est intéressant de remarquer qu'il peut être impossible de passer d'un état valide à un autre état valide sans passer par un état qui falsifierait une contrainte. C'est pourquoi, nous définirons la notion de transaction qui est une séquence indivisible eu égard aux contraintes d'intégrité. On pourra faire l'analogie avec les concepts d'actions atomiques et d'invariants en programmation parallèle. Ainsi, prouver qu'une transaction vérifie une contrainte d'intégrité est équivalent à prouver qu'une action atomique préserve un invariant.

Enfin, le composant intégrité du modèle relationnel est composé de deux règles d'intégrité, l'intégrité des identifiants et l'intégrité référentielle [COOD 82] .

L'intégrité des identifiants est une contrainte qui assure qu'aucune valeur d'identifiant primaire ne peut être nulle. La raison en est que dans le modèle relationnel, il est possible de distinguer tous les tuples d'une relation et que dès lors il existe un identifiant pour cette relation tel que chaque tuple ait une identification unique. Les identifiants primaires assurent la fonction d'identification dans les bases de données relationnelles et ce serait contredire cette propriété que de permettre qu'une valeur d'identifiant primaire soit totalement ou partiellement nulle.

La règle d'intégrité référentielle est quelque peu plus compliquée. Schématiquement, cette contrainte nous assure que tout composant qui prend sa valeur sur un domaine qui admet un identifiant primaire, est soit une valeur nulle soit une valeur identique à celle d'un identifiant primaire. Elle nécessite la définition suivante

définition : On appelle domaine primaire, un domaine sur lequel on a défini un identifiant primaire à une seule valeur d'attribut.

La contrainte peut alors s'exprimer de la manière suivante

Soit D un domaine primaire.
Soit R_1 une relation avec un attribut de nom A défini sur D.
A tout instant, chaque valeur des composants des tuples de R_1 correspondant à l'attribut A doit être
- soit nulle

- soit égale à v où v est la valeur d'identifiant primaire d'un certain tuple dans une relation R_2 dont l'identifiant primaire est définie sur D.

Nous remarquerons que rien n'impose à R_1 et à R_2 d'être distinctes et que par définition de domaine primaire la relation R_2 doit exister.

PARTIE II : CONTRIBUTION DE LA LOGIQUE DES PREDICATS DU PREMIER ORDRE AUX BASES DE DONNEES RELATIONNELLES.

CHAPITRE 1. FORMALISATION D'UNE BASE DE DONNEES RELATIONNELLE. [GALL 84]

1.0 Introduction.

Ce chapitre traite de la formalisation d'une base de données relationnelle en logique des prédicats du premier ordre. Afin de mener à bien cette formalisation, les hypothèses sous-jacentes aux bases de données relationnelles seront mises en évidence. Nous présenterons donc, l'hypothèse de fermeture du domaine, l'hypothèse du monde fermé, et l'hypothèse d'unicité des noms. Nous introduirons ensuite les deux formalisations : la formalisation selon la vue "théorie du modèle" qui considère la base de données comme une interprétation d'un langage du premier ordre et la formulation selon la vue "théorie de la preuve" qui considère la base de données comme une théorie du premier ordre.

1.1 Hypothèses des bases de données relationnelles.

Dans ce paragraphe, nous examinerons successivement la représentation des informations dans le modèle relationnel et les différentes hypothèses sous-jacentes à cette représentation.

1.1.1 La représentation des informations.

Une base de données a pour objectif de représenter et de maintenir une connaissance sur une certaine réalité appelée le monde modélisé. Dans le cadre des bases de données relationnelles, cette connaissance peut être représentée par un ensemble d'éléments (individus) reliés entre eux par des relations et des fonctions. Une fonction $y = f(x)$ peut être représentée par une relation $f(x,y)$ et une contrainte $\forall x \forall y \forall z$ $F(x,y) \wedge F(x,z) \rightarrow (y=z)$. C'est pourquoi, nous nous limitons aux relations. Cette connaissance, représentée sous forme d'ensembles d'éléments reliés entre eux par des relations, peut contenir des informations de deux types : les informations élémentaires et les informations générales

- les informations élémentaires (ou faits élémentaires) sont de la forme "Pierre est le père de Paul .
- les informations générales ou lois générales sont de la forme
  "Le père d'un père est un grand-père".
  "Tout individu n'a qu'un père".

Ces informations, élémentaires ou générales, peuvent être représentées par des formules bien formées. En effet, on peut associer à chaque

relation d'arité n, un symbole de prédicat n-aire et à chaque élément d'un domaine, une constante.
Ainsi, pour les informations définies ci-dessus, nous aurons :

- père (Pierre, Paul)
- $\forall x \forall y \forall z$ (père (x,z) $\wedge$ père (z,y)) $\rightarrow$ grand-père (x,y)
- $\forall x \forall y \forall z$ (père (y,x) $\wedge$ père (z,x)) $\rightarrow$ (y = z)

Dans une base données relationnelle, seuls les faits élémentaires ou informations élémentaires sont représentées. Les lois générales sont utilisées comme contraintes d'intégrité. Les informations élémentaires constituent alors un état de base de données et les contraintes d'intégrité délimitent les états valides de la base de données.

De même, la connaissance que nous désirons représenter, peut contenir des informations négatives aussi bien que des informations positives. Si l'on conçoit bien comment représenter les informations positives, il est intéressant de s'attarder un peu à la représentation des informations négatives.

Les informations négatives peuvent en fait, être représentées de trois manières distinctes :

1° Elles peuvent être représentées explicitement. L'information négative est alors représentée par un littéral négatif.
$\neg R(x_1, x_2, \ldots, x_n)$ où $x_1 \ldots x_n$ sont des constantes.

2° Nous pouvons représenter les informations négatives implicitement par l'expression de lois générales qui permettent de les déduire. Nous considérons à cet égard, deux types de lois.

- <u>des lois d'unicité.</u>

  age(Peter,35) peut être déduit du fait age(Peter,36) et de la loi générale exprimant que tout individu a un et un seul age.

- <u>des lois de complétude.</u>

  Ces lois expriment que pour une relation spécifiée, tout fait qui n'est pas mentionné dans la relation est faux. Cette convention a pour effet de fusionner les faits faux et les faits inconnus. Par exemple, la loi générale : $\forall x(P(x) \rightarrow (x=a) \vee (x=b))$
  est une loi de complétude qui établit que tout fait P(c) avec $c \neq a$ et $c \neq b$ est faux.

Nous pouvons présenter maintenant les hypothèses prises en compte dans les bases de données relationnelles. La première est une convention pour les informations négatives. La seconde concerne la fermeture du domaine et la troisième l'unicité des noms.

1.1.2 L'hypothèse du monde fermé.

Pour assurer la réponse aux questions contenant des négations, deux hypothèses, l'hypothèse du monde fermé (Closed World Assumption ou CWA) et l'hypothèse du monde ouvert (Open World Assumption ou OWA) peuvent être considérées. Les bases de données relationnelles utilisent la CWA mais nous allons présenter les deux hypothèses pour bien percevoir leurs différences.

L'hypothèse du monde ouvert correspond à l'approche traditionnelle de la logique du premier ordre pour l'évaluation des questions. Sous OWA, les seules informations tenues pour vraies sont celles qui se trouvent explicitement dans la base de données. Considérons les informations élémentaires suivantes :

professeur (a)	élève (A)	enseigne (a,A)
professeur (b)	élève (B)	enseigne (a,B)
professeur (c)	élève (C)	enseigne (b,B)
professeur (d)		enseigne (c,C)

où professeur (x) exprime que x est un professeur
élève (x) exprime que x est un élève
enseigne (x,y) exprime que le professeur x enseigne à l'élève y.

Considérons la question :
"Quels sont les professeurs qui n'enseignent pas à B ?"

$$\{ y \mid \text{professeur}(y) \wedge \neg\, \text{enseigne}(y,B) \}$$

La réponse sous OWA sera l'ensemble vide. Cette réponse peut paraître peu intuitive car nous aurions désiré avoir $\{c,d\}$. Cela est dû au fait que la logique du premier ordre ne tient pour certain que ce qui est représenté. Ce n'est pas parce que le tuple $<c,B>$ ne figure pas dans la relation enseigne que l'on peut en déduire $\neg$ enseigne (c,B). Pour obtenir comme réponse l'ensemble $\{c,d\}$, nous aurions dû représenter explicitement $\neg$ enseigne(c,B) et $\neg$ enseigne(d,B).

L'hypothèse du monde fermé suppose, quant à elle, une connaissance totale du monde modélisé. Comme cette connaissance est totale, seuls les faits positifs doivent être effectivement représentés. Un fait négatif est implicitement présent si son homologue positif ne l'est pas. Sous CWA, la réponse à l'exemple et la question précédente aurait été $\{ c,d \}$.

Sous CWA, pour toute relation $R(x_1,x_2,\ldots,x_n)$ et pour tout tuple $\langle a_1,a_2,\ldots,a_n \rangle$ soit $R(a_1,a_2,\ldots,a_n)$ soit $\neg R(a_1,a_2,\ldots,a_n)$ est vrai. Nous noterons que l'hypothèse du monde fermé a pour conséquence de fusionner les faits faux et les faits inconnus lorsque la connaissance du monde n'est pas totale.

1.1.3 L'hypothèse de fermeture du domaine.

Cette hypothèse établit qu'il n'y a pas d'autres individus que ceux qui sont présents dans la base de données. Cette hypothèse n'est pas nécessaire lorsque nous ne considérons que des questions existentiellement quantifiées. Les réponse à de telles questions sont identiques, et cela a été prouvé par Reiter dans [REIT 78], avec ou sans cette hypothèse.

Par contre, cette hypothèse prend toute son importance lors de la réponse à des questions universellement quantifiées. Pour illustrer la nécessité de cette hypothèse, nous allons présenter deux exemples.

Considérons la base de données :

professeur (a)	élève (A)	enseigne (a,A)
professeur (b)	élève (B)	enseigne (a,B)
	élève (C)	enseigne (a,C)
		enseigne (a,D)

Considérons la question :

"Quels sont les professeurs qui enseignent à tous les élèves ?"

La réponse que nous attendons est $\{ a \}$. Cependant, en l'absence de l'hypothèse de fermeture du domaine, nous n'aurons aucune preuve de

$$\forall y \text{ élève}(y) \rightarrow \text{enseigne}(a,y) \qquad (1)$$

La raison en est qu'il peut exister des élèves qui ne sont pas dans la base de données **et** auxquels 'a' n'enseigne pas c-à-d qu'il peut exister des états de la base de données qui contiennent des individus supplémentaires qui sont des élèves et auxquels 'a' n'enseigne pas.

Le problème réside dans la signification du "Pour tout" [REIT 80]. La réponse 'a' ne sera valable dans tous les états de la base de données

que si nous avons une loi générale de la forme (1). Cette loi garantit que 'a' est une réponse valable quel que soit le domaine d'interprétation.

Ces considérations mettent en évidence la véritable sémantique que nous associons au "Pour tout" dans la question. Ce qui nous intéresse ici ce n'est pas d'avoir une réponse qui soit vraie dans tous les états de la base de données mais bien une réponse qui soit vraie dans l'état actuel de la base de données. La sémantique du "Pour tout" est donc "Pour tout élève dont on connaît l'existence".

Un deuxième exemple d'utilisation de l'axiome de fermeture est une question du genre : "Quels sont les individus $x_1, x_2, \ldots, x_n$ telle que $\langle x_1, x_2, \ldots, x_n \rangle \notin$ R où R est une relation". Telle qu'elle est énoncée, cette question a peu de sens. Il est en effet impensable de lister tous les tuples qui n'appartiennent pas à R. Cette liste sera d'ailleurs plus que probablement infine. D'autre part, sur quel domaine les $x_1, \ldots x_n$ vont-ils prendre leurs valeurs ? L'hypothèse de fermeture du domaine délimite dans chacun des deux cas, l'ensemble des individus à prendre en considération.

La prise en compte de l'hypothèse de fermeture du domaine est implicite dans les opérations de l'algèbre relationnelle. Dans le cas du calcul relationnel, cette hypothèse sera remplacée par une convention qui consiste à restreindre la forme des questions à des formules saines qui sont des questions qui restreignent d'elles-mêmes le domaine sur lequel les variables prennent leurs valeurs ou par l'utilisation de types.

1.1.4 L'hypothèse d'unicité des noms.

Cette hypothèse établit simplement que deux individus qui ont des noms distincts sont différents. Par exemple, a et b représentent nécessairement des professeurs différents.

1.2 Formalisation d'une base de données selon la vue "théorie du modèle".

Cette formalisation consiste à considérer une base de données comme l'interprétation d'une théorie du premier ordre.

Pour ce faire, il convient de définir un langage logique de premier ordre L. Considérons une base de données DB ainsi que le domaine D qui est l'union de tous les éléments des domaines de la base de données.

Le langage du premier ordre contiendra un ensemble de constantes qui correspondent chacune à un élément de D et à tout élément de D sera associé une constante. Le langage contiendra aussi un symbole de prédicat n-aire R' pour chaque relation R de la base de données. Le langage peut ensuite être augmenté pour prendre en compte les opérations de comparaison ($>$, $<$, $\leq$, $\geq$, $=$, $+$) auxquels on assigne leur signification habituelle. Le langage est supposé n'avoir aucun symbole de fonction.

La base de données peut alors être considére comme une interprétation du langage L de domaine D et à laquelle correspondent les assignations énoncées ci-dessus.
De plus, si on définit un certain nombre de formules bien formées sur ce langage, la base de données peut être considérée comme une interprétation de la théorie constituée de ces formules.
Dans cette interprétation, une formule bien formée est évaluée selon la définition que nous avons donné dans la section 1 de la première partie. Ainsi, la formule atomique $R'(e'_1, e'_2, \ldots, e'_n)$ où $e'_1, e'_2, \ldots, e'_n$ sont des constantes est vraie si $\langle e_1, e_2, \ldots, e_n \rangle \in R$ où $e_1, \ldots, e_n$ sont les valeurs assignées à $e'_1, e'_2, \ldots, e'_n$. Sinon elle est fausse.

Les questions seront formulées comme des formules bien formées. On distingue deux formes de questions. Les questions formées qui demandent une réponse par 'oui' ou par 'non'. Elles seront représentées par des formules bien formées fermées. Les questions ouvertes demandent comme réponse un ensemble de tuples. Nous prendrons comme convention de représenter une question ouverte par une formule bien formée qui contient des variables libres. Soit $F(x_1, \ldots, x_n)$ une formule bien formée qui contient $x_1, x_2, \ldots, x_n$ comme variables libres. Cette question demande l'ensemble des tuples $\langle e_1, e_2, \ldots, e_n \rangle$ tels que $F(e_1, e_2, \ldots, e_n)$ est vrai.

L'évaluation des questions dans cette approche est la suivante. La réponse à une question fermée est la valeur de vérité assignée à cette question dans l'interprétation. Soit une question W qui contient n variables libres. L'évaluation de cette question donne comme réponse l'ensemble des tuples $\langle e_1^{(1)}, \ldots, e_m^{(1)} \rangle, \ldots, \langle e_1^{(n)}, \ldots, e_m^{(n)} \rangle$ qui donne la valeur "vrai" à la formule W.
Nous serons heureux de remarquer que cette évaluation se fait naturellement sous les hypothèses de fermeture du domaine, du monde fermé et d'unicité des noms. En effet, on se place sous l'hypothèse de fermeture

du domaine puisque le domaine d'interprétation est constitué de tous les éléments des domaines. De même, l'hypothèse du monde fermé est implicite puisque, dans une interprétation, ¬ A est vrai si A est faux.

Les contraintes d'intégrité peuvent être exprimées comme des formules bien formées fermées du langage L. Soit Ic l'ensemble des contraintes d'intégrité de la base de données. Nous dirons que la base de données est dans un état valide si et seulement si toutes les contraintes Ic se voient attribuer la valeur vraie dans l'interprétation c-à-d si l'interprétation est un modèle de Ic.

Cette approche est l'approche qui est habituellement suivie dans les bases de données. Nous présenterons dans la suite l'apport de cette formalisation pour les langages d'interrogation et les contraintes d'intégrité.
Cependant, cette approche se prête mal à la prise en compte de lois générales pour la représentation des informations.
En effet, si des lois générales sont exprimées, elles ne le sont que dans la théorie dont la base de données est une interprétation. Ces règles générales sont supposées être toujours vraies et la base de données devrait en constituer un modèle. Les lois générales sont donc des contraintes d'intégrité.

1.3 Formalisation de la base de données selon la vue "théorie de la preuve".

Cette formalisation consiste à formaliser la base de données comme une théorie du premier ordre. Elle s'inspire de l'approche suivie dans les systèmes questions-réponses, excepté que ces systèmes admettent l'expression de lois générales, ce qui n'est pas le cas des bases de données relationnnelles.

Cette formalisation se base également sur le langage de premier ordre L défini ci-dessus. Elle consiste à construire sur ce langage une théorie T qui admet la base de données comme unique modèle (à un isomorphisme près). La théorie T sera construite à partir des axiomes suivants.

1. Les faits élémentaires : Pour toute relation dans la base de données R et pour tout tuple $\langle a_1, \ldots, a_n \rangle \in R$, l'axiome $R'(a'_1, \ldots, a'_n) \in T$.

2. Les axiomes de particularisation.

(1) les axiomes de complétude

Pour chaque relation R dans la base de données, nous construisons un axiome de complétude. Soit $\langle e_1^{(1)}, e_2^{(1)}, \ldots, e_n^{(1)} \rangle, \ldots, \langle e_1^{(t)}, e_2^{(t)}, \ldots, e_n^{(t)} \rangle$, les tuples dans la relation R. L'axiome de complétude pour R est défini par

$$\forall x_1 \ldots x_n \; R'(x_1, \ldots, x_n) \rightarrow ((x_1 = e_1^{(1)'} \wedge \ldots \wedge x_n = e_n^{(1)'}) \vee \ldots \vee (x_1 = e_1^{(t)'} \wedge \ldots \wedge x_n = e_n^{(t)'})).$$

Cet axiome à pour effet d'établir que les seuls tuples qui satisfont R sont $\langle e_1^{(1)}, \ldots, e_n^{(n)} \rangle, \ldots, \langle e_1^{(t)}, \ldots, e_n^{(t)} \rangle$ et est donc la prise en considération de l'hypothèse du monde fermé.

(2) les axiomes d'unicité de noms.

Si $e_1, e_2, \ldots, e_q$ sont tous les individus dans la base de données, les axiomes d'unicité des noms sont :

$$\neg(e_1' = e_2'), \ldots, \neg(e_1' = e_2'), \neg(e_2' = e_3'), \ldots, \neg(e_{q-1}' = e_q')$$

(3) l'axiome de fermeture du domaine.

$$\forall x \quad (x = e_1') \vee \ldots \vee (x = e_q')$$

Ces axiomes ont pour effet de représenter les hypothèses de fermeture du domaine et d'unicité des noms.

(4) les axiomes d'égalité.

Les axiomes d'égalité sont rendus nécessaires par la présence des symboles d'égalité dans les axiomes précédents. Les axiomes d'égalité sont

- réflexivité : $\forall x \; (x = x)$
- symétrie : $\forall x \forall y \; ((x = y) \rightarrow (y = x))$
- transitivité : $\forall x \forall y \forall z \; ((x = y) \wedge (y = z) \rightarrow (x = z))$
- principe de substitution des termes égaux

$$\forall x_1 \forall x_2 \ldots \forall x_n \forall y_1 \forall y_2 \ldots \forall y_n \; (P(x_1, \ldots, x_n) \wedge (x_1 = y_1) \wedge \ldots \wedge (x_n = y_n) \rightarrow P(y_1, \ldots, y_n)).$$

On peut montrer que cette théorie admet la base de données comme unique modèle (à un isomorphisme près) [GALL 84a].

Les questions sont formulées de la même manière que dans la formali-

sation selon la théorie de la preuve. Leur évaluation est cependant différente.
Soit une question fermée F. La réponse à cette question est "oui".SI T $\vdash$ F. Elle est égale à 'non' si T $\nvdash$ F. Le problème est décidable puisque le nombre de constantes est fini et qu'il n'y a pas de symboles de fonction.
Soit une question ouverte $F(x_1, x_2, \ldots, x_n)$ où $x_1, x_2, \ldots, x_n$ sont les variables libres de F. La réponse à cette question est l'ensemble des tuples $\langle e_1, e_2, \ldots, e_n \rangle$ tels que $T \vdash F(e_1, e_2, \ldots, e_n)$ où $F(e_1, \ldots, e_n)$ est la formule F où on a remplacé les variables $x_1 \ldots x_n$ par les constantes $e_1 \ldots e_n$.
Autrement dit, cela revient à trouver toutes les preuves de $\exists\, x_1 \ldots x_n$ $F(x_1, \ldots, x_n)$.

Les contraintes d'intégrité peuvent être représentées par des formules bien formées fermées. Si $\mathcal{S}$ est l'ensemble des contraintes, nous dirons que la base de données est dans un état valide si pour chaque contrainte $\psi \in \mathcal{S}$ nous avons $T \vdash \psi$.
Kowalski [KOWA 79] définit la vérification des contraintes d'intégrité d'une manière différente. La base de données satisfait les contraintes d'intégrité Ic si et seulement si Ic $\cup$ T est une théorie cohérente.

Cette formalisation est peu adéquate pour la formalisation d'une base de données. En effet, les hypothèses sont, cette fois, représentées explicitement via les axiomes de particularisation. Cependant, son principal intérêt réside dans les généralisations qu'elle suggère. Si nous ajoutons des faits disjonctifs, nous obtenons une base de données avec des informations incertaines. Un exemple d'information incertaine est $P(a) \vee Q(a)$. En effet, cette formule nous dit que $P(a) \vee Q(a)$ est vrai c-à-d que P(a) est vrai ou que Q(a) est vrai ou encore que P(a) et Q(a) sont vrais mais nous n'avons pas suffisamment d'informations pour décider laquelle de ces deux alternatives est vraie.
De même, si nous ajoutons comme axiomes des lois générales, nous obtenons une base de données déductive .
Ces deux extensions seront envisagées au chapitre "Base de données déductives".

1.4 Conclusion.

Nous avons présenté, dans ce chapitre, les différentes hypothèses prises en compte implicitement dans les bases de données relationnelles, à savoir l'hypothèse de fermeture du domaine, l'hypothèse du monde fermé et l'hypothèse d'unicité des noms. Ensuite, nous avons présenté deux formalisations d'une base de données relationnelle : la formalisation selon la vue 'théorie du modèle' qui considère une base de données relationnelle comme une interprétation d'un langage du premier ordre et la formalisation selon la vue 'théorie de la preuve' qui considère une base de données comme une théorie du premier ordre. La première formalisation est celle implicitement suivie dans les bases de données. Nous verrons sa contribution pour les composants manipulation et intégrité dans les chapitres 3 et 4. La formalisation selon la vue 'théorie de la preuve' sera exploitée dans le chapitre qui suit "Les bases de données déductives".

CHAPITRE 2. BASES DE DONNEES DEDUCTIVES.

2.0 Introduction.

Dans ce chapitre, nous étudierons la contribution de la logique des prédicats du premier ordre, par l'intermédiaire de la formalisation selon la vue"théorie de la preuve" ou composant structuration du modèle relationnel. Cette formalisation suggère une généralisation du composant structuration par la prise en compte de lois générales débouchant par là, sur les bases de données déductives. Très schématiquement, une base de données déductive est une base de données où de nouveaux faits peuvent être déduits des faits qui sont introduits explicitement. L'origine de ces bases de données doit être trouvée dans deux champs : le champ "bases de données" qui n'a pas manqué de remarquer qu'une base de données relationnelle avait de nombreuses ressemblances avec la logique des prédicats du premier ordre et le champ "programmation en logique" qui s'est aperçu que la forme clausale avait de fortes similitudes avec les relations du modèle relationnel. Ce rapprochement a donné lieu, selon le champ que l'on considère, aux bases de données déductives (champ"bases de données") et aux bases de données logiques (champ"programmation en logique"). Ce chapitre sera essentiellement divisé en deux parties : la partie "représentation des informations" et la partie"mise en oeuvre des règles de déduction". La première définira deux formes de bases de données déductives selon la forme des clauses acceptées comme règle de déduction : les bases de données déductives définies et les bases de données déductives indéfinies Les bases de données déductives définies ne considèrent que des clauses définies (un et un seul littéral positif) comme règles de déduction tandis que les bases de données déductives indéfinies acceptent plus d'un littéral positif. Nous donnerons deux définitions d'une base de données déductives définies : une définition en logique du premier ordre et une définition opérationnelle. La définition opérationnelle fera apparaitre une nouvelle règle d'inférence : C W A. Nous montrerons que cette règle ne peut être utilisée comme telle dans les bases de données déductives indéfinies et nous en donnerons une généralisation. Nous dirons enfin quelques mots sur les valeurs nulles dans les bases de données déductives indéfinies et sur la séparation des lois générales en règles de déduction et contraintes d'intégrité.
La seconde partie "mise en oeuvre des règles de déduction" se préoccup-

pera de la mise en oeuvre effective des bases de données déductives définies. Elle présentera les deux grandes approches : l'approche en génération et l'approche en dérivation. L'approche en génération utilise les règles de déduction pour saturer la base de données de tous les faits déductibles, tandis que l'approche en dérivation déduit les faits lorsque cela s'avère nécessaire, par exemple en réponse à une question. L'approche en dérivation a donné lieu à deux méthodes, la méthode compilée et la méthode interprétée.
Nous terminerons ce chapitre par une réflexion sur les bases de données déductives.

2.1 Représentation des informations.

2.1.1 Prise en compte des lois générales dans les deux formalisations.

Une base de données déductive permet de représenter des informations générales ou des lois générales. Comment ces lois sont-elles prises en compte dans les deux formalisations ?

Selon la formalisation "théorie de la preuve", toutes les lois générales sont prises en compte comme des axiomes propres de la théorie. Les informations implicites (informations qui peuvent être déduites des informations élémentaires et des lois générales) sont des théorèmes de cette théorie.

Selon la formalisation "théorie du modèle ", toutes les lois générales sont utilisées en tant que contraintes d'intégrité. La base de données est un modèle de ces lois générales. Leur seul effet est de réduire les états valides de la base de données.

Chacune de ces deux approches fait donc un usage excusif des lois générales, soit en tant que règles de déduction, soit en tant que contraintes d'intégrité. Or, il s'avère que certaines lois générales ont une plus grande utilité en tant que contraintes d'intégrité et d'autres en tant que règles de déduction. En effet, la loi générale exprimée sous forme de clauses de Horn

grand-père (x,y) ← père (x,z) ∧ père (z,y).

exprimant que x est le grand-père de y si x est le père de z et z le père de y, est plus appropriée en tant que règle de déduction. Elle permet de déduire à partir de la relation "père" un certain nombre de tuples qui appartiendront à la relation "grand-père".

Par contre, la loi générale

(âge < 150) ← individu(nom, âge, adresse)

exprimant que l'âge d'un individu ne peut excéder 149 ans, est plus appropriée comme contrainte d'intégrité. Utilisée en tant que règle de déduction, elle ne génèrerait que des faits redondants du type "21 < 150" ou des faits faux de la forme "160 > 150".
Par conséquent, un usage systématique des lois générales, soit en règles de déduction soit en contraintes d'intégrité, paraît peu adéquat. L'approche qui sera prise en compte dans les pages qui suivent consistera à traiter les lois générales en partie comme règle de déduction et en partie comme contraintes d'intégrité. Nous donnerons également des critères qui nous guideront pour réaliser le partage des lois générales.

2.1.2 Définition d'une base de données déductive définie.

Dans le cadre des bases de données déductives définies, les règles de déduction seront des clauses définies c-à-d des clauses qui contiendront toujours un et un seul littéral positif. On remarquera que les clauses définies sont également les clauses utilisées pour définir un programme logique. La motivation derrière ce choix est un problème de performances. Nous verrons que la mise en oeuvre des bases de données déductives définies peut se faire de manière relativement efficace, ce qui ne sera pas le cas des bases de données déductives indéfinies.

Dans ce qui suit, nous donnerons une définition d'une base de données déductive définie en logique du premier ordre. Cependant, les axiomes de particularisation font en sorte que la théorie d'une base de données déductive définie ne pourra être constituée d'un ensemble de clauses définies. Nous donnerons alors une définition opérationnelle qui ramènera la théorie de la base de données à un ensemble de clauses définies.

2.1.2.1 Définition d'une base de données déductive définie en logique du premier ordre.

Une base de données déductive définie est caractérisée par

(1) une théorie du premier ordre T dont les axiomes propres sont

a) les axiomes de particularisation : l'axiome de fermeture de domaine, les axiomes d'unicité des noms, les axiomes de complétude et les axiomes d'égalité.

b) les faits élémentaires : représentés par des formules de la forme $P(t_1, t_2, \ldots, t_n) \leftarrow$ où $t_1, t_2, \ldots, t_n$ sont des constantes.

c) les règles de déduction : ces règles sont des clauses définies sans symboles de fonction.

(2) un ensemble de contraintes d'intégrité IC qui sont des formules bien formées fermées.

Les symboles de fonction sont exclus pour obtenir des réponses définies et explicites aux questions soumises à la base de données.
Les axiomes de complétude définis ci-dessus sont une généralisation des axiomes de complétude définis dans la formalisation selon la vue "théorie de la preuve". Contrairement à cette dernière où les axiomes de complétude ne concernaient que les faits élémentaires, ils devront également considérer les règles de déduction. En effet, les règles de déduction n'expriment que la partie "si" d'une définition si et seulement si ". Par conséquent, on ne peut déduire que des informations positives de ces règles. Si l'on veut pouvoir déduire des informations négatives, il faudra d'une manière ou d'une autre, exprimer la partie "seulement si " de la définition. Le but des axiomes de complétude est justement l'expression de cette partie "seulement si".
Schématiquement, pour construire un axiome de complétude pour un symbole de prédicat n-aire P, nous considérons l'ensemble C de toutes les clauses qui ont ce symbole de prédicat comme conséquent. Nous formerons une clause dont l'antécédent est P et où le conséquent est une disjonction des antécédents des clauses de C.

Examinons maintenant précisément comment il est possible de construire ces axiomes.
Considérons une clause

$$P(t_1, t_2, \ldots t_n) \leftarrow L_1 \wedge L_2 \wedge \ldots \wedge L_m \quad (m \geqslant 0)$$

qui est soit un fait élémentaire, soit une règle de déduction.
Soit $x_1, x_2, \ldots, x_n$ des variables qui n'apparaissent pas dans la clause
Nous pouvons la transformer en

$$P(x_1, x_2, \ldots, x_n) \leftarrow (x_1 = t_1) \wedge \ldots \wedge (x_n = t_n) \wedge L_1 \wedge \ldots \wedge L_m \quad (2)$$

Si $y_1, y_2, \ldots, y_d$ sont les variables de la clause initiale, nous pouvons transformer (2) en

$$P(x_1, x_2, \ldots, x_n) \leftarrow \exists y_1 \ldots \exists y_d ((x_1 = t_1) \wedge \ldots \wedge (x_n = t_n) \wedge L_1 \wedge \ldots \wedge L_m)$$

Supposons maintenant qu'une transformation semblable ait été réalisée pour toutes les clauses qui admettent P comme conclusion. Nous aurons k ($\geqslant 1$) clauses de la forme

$P(x_1, x_2, \ldots, x_n) \leftarrow \epsilon_1$

$P(x_1, x_2, \ldots, x_n) \leftarrow \epsilon_k$

où chaque ϵ_i $(1 \leqslant i \leqslant k)$ sera de la forme

$$\exists y_1 \exists y_2 \ldots \exists y_d \; ((x_1 = t_1) \wedge \ldots \wedge (x_n = t_n) \wedge L_1 \wedge \ldots \wedge L_m).$$

L'axiome de complétude sera défini par

$$\forall x_1 \; \forall x_2 \ldots \forall x_n \; (P(x_1, x_2, \ldots, x_n) \rightarrow \epsilon_1 \vee \epsilon_2 \vee \ldots \vee \epsilon_k).$$

D'autre part, certains prédicats définis dans le langage peuvent ne pas apparaitre comme conclusion d'une clause définie. Par conséquent, pour tout prédicat q dans ce cas, nous définirons un axiome de complétude

$$\forall x_1 \; \forall x_2 \ldots \forall x_n \; \neg q(x_1, \ldots, x_n).$$

Il devrait être clair que la base de données ainsi définie n'est pas constituée d'un ensemble de clauses définies.

Les questions et leur évaluation peuvent être définies comme pour la formalisation d'une base de données relationnelle selon la vue "théorie de la preuve". Il en est de même des contraintes d'intégrité.

2.1.2.2 Définition opérationnelle d'une base de données déductive définie.

Une base de données déductive définie ne sera jamais mise en oeuvre en suivant la définition qui vient d'être présentée. Elle serait par trop innefficace en raison des axiomes de particularisation. Cependant, il est possible de donner une autre définition qui permettra de ramener les axiomes de la théorie de la base de données à des clauses définies. Cette définition différera de la première en ce sens que les axiomes de particularisation seront remplacés par des restrictions sur les questions, les contraintes et les règles de déduction et une nouvelle règle d'inférence C W A. Nous examinerons successivement comment on peut supprimer l'axiome de fermeture du domaine et les axiomes de complétude.

2.1.2.2.1 L'axiome de fermeture du domaine.

Dans un premier temps, il est possible de supprimer l'axiome de fermeture du domaine en ne considérant que des formules à champ restreint pour les questions, les règles de déduction et les contraintes d'intégrité. Les formules à champ restreint sont un sous-ensemble des formules saines [ULLM 80] que nous étudierons au chapitre suivant.

Une formule sous forme normale prenexe conjonctive est à champ restreint si les deux conditions suivantes sont vérifiées [NICO 79].

(1) chacune de ces variables universellement quantifiées apparait dans au moins un littéral négatif des disjonctions qui contiennent cette variable.

(2) pour chaque variable existentiellement quantifiée qui apparait dans un littéral négatif, il existe une disjonction contenant seulement des littéraux positifs qui contiennent tous cette variable comme argument.

Les questions, les contraintes et les règles de déduction de mise à jour devront satisfaire ces conditions. La définition peut, cependant, s'exprimer de manière plus simple pour les règles de déduction. En effet, si nous considérons des formules sous forme de clauses, la propriété du champ restreint peut s'exprimer par

une clause $P_1 \vee \ldots \vee P_m \leftarrow q_1 \wedge q_2 \wedge \ldots \wedge q_n$ est à champ restreint si toute variable apparaissant dans la conclusion apparait également dans le corps de la clause.

2.1.2.2.2 L'hypothèse du monde fermé.

Les axiomes de complétude et d'unicité des noms peuvent être remplacés par une règle d'inférence C.W.A. Dans ce qui suit, nous examinerons la définition sémantique et syntaxique de cette règle d'inférence. La définition sémantique mettra en évidence pourquoi on ne peut déduire des informations négatives d'un ensemble de clauses définies. Elle permettra également de montrer ultérieurement pourquoi la règle d'inférence C W A ne peut être utilisée dans les bases de données déductives indéfinies. Nous noterons également que, comme les axiomes d'égalité ne sont nécessaires que parce que le symbole d'égalité apparaissait dans les axiomes de complétude, d'unicité des noms et de fermeture du domaine, ceux-ci ne seront plus nécessaires.

2.1.2.2.2.1 Définition sémantique de C.W.A. [LLOY 84]

Lorsque nous considérons des formules exprimées sous forme de clauses, nous pouvons, au lieu de considérer la notion générale d'interprétation, nous restreindre à une classe d'interprétations, les interprétations de Herbrand, afin de définir la sémantique des formules. Ces interprétations de Herbrand

sont également utilisées pour définir la sémantique d'un langage de programmation logique [KOWA 76]. Ces interprétations permettront de donner une définition sémantique de la règle d'inférence C W A.

Interprétation de Herbrand.

Considérons L un langage de premier ordre pour l'expression de clauses. L'univers de Herbrand U_1 pour L est l'ensemble de tous les termes de base (ie : qui ne contiennent pas de variables) qui peuvent être formés à partir des constantes et des symboles de fonction qui apparaissent dans L. Dans le cas où L n'a pas de constantes, il sera nécessaire d'en ajouter une dans le but de former les termes de base.
La base de Herbrand B_1 pour le langage L est l'ensemble de toutes les formules atomiques $P(t_1, t_2, \ldots, t_n)$ où P est un symbole de prédicat n-aire de L et où $t_1, t_2, \ldots, t_n$ sont des termes appartenant à l'univers d'Herbrand U_1.
Une interprétation de L est une interprétation de Herbrand si les conditions suivantes sont vérifiées

(1) le domaine d'interprétation est l'univers de Herbrand U_1
(2) les constantes dans L sont assignées à elles-mêmes
(3) si f est un symbole de fonction n-aire de L, alors f est assignée à la fonction de $(U_1)^n$ dans U_1 définie par $(t_1, t_2, \ldots, t_n) \rightarrow f(t_1, t_2, \ldots, t_n)$

Comme précédemment, nous parlerons, par abus de langage, d'une interprétation d'un ensemble de formules bien formée S de L plutôt que d'une interprétation du langage L. On parlera alors de l'univers de Herbrand U_s et de la base de Herbrand B_s d'un ensemble de formules S.
Nous remarquerons, par ailleurs, qu'aucune assignation n'a été définie pour les symboles de prédicat. C'est pourquoi, différentes assignations définiront différentes interprétations.
Comme les constantes et les fonctions ont une assignation fixe, une interprétation de Herbrand pour un ensemble de clauses S est un sous-ensemble de la base de Herbrand B_s. Par conséquent, pour un symbole de prédicat n -aire P et des termes $t_1, t_2, \ldots t_n$, $P(t_1, \ldots, t_n)$ est vrai dans une interprétation si et seulement si

$P(t_1, t_2, \ldots, t_n) \in I$.

L'univers de Herbrand sera infini si l'ensemble des formules admet au moins un symbole de fonction. Sinon il sera fini. Il en est évidemment de même pour la base de Herbrand B_s.

Modèle de Herbrand.

Soit I une interprétation de Herbrand.
Une formule atomique de base A est vraie dans I ssi $A \in I$.
Un littéral négatif $\neg A$ est vrai dans I ssi $A \notin I$.
Une clause de base $L_1 \vee L_2 \vee \ldots \vee L_m$ est vraie dans I si au moins un littéral L_i $(1 \leqslant 1 \leqslant m)$ est vrai dans I.
Une clause C est vraie dans I ssi $C\sigma$ est vraie dans I pour toute substitution σ telle que $C\sigma$ est une clause de base.
Un ensemble de clauses A est vrai dans I ssi chaque clause de A est vraie dans I.
Nous dirons que I est un modèle de Herbrand de A ssi A est vrai dans I. De même, on peut montrer [KOWA 76] que si A admet un modèle alors il admet un modèle de Herbrand. D'autre part, si nous nous limitons à un ensemble de clauses de Horn A et si m(A) est un ensemble de modèles de Herbrand pour A, alors l'intersection des modèles de A, notée $\bigcap m(A)$, est encore un modèle de Herbrand de A [KOWA 76] .
Si nous nous restreignons à un ensemble de clauses définies A alors l'ensemble des modèles de Herbrand de A est non vide. En effet, cet ensemble contient B_A. Considérons, en effet, une interprétation I d'un ensemble de clauses définies A. Cette interprétation ne sera pas un modèle si et seulement si il existe une clause de la forme

$$A_o \vee \neg A_1 \vee \ldots \vee \neg A_n$$

telle que $A_1, A_2, \ldots, A_n \in I$ et $A_o \notin I$. Si l'interprétation est la base de Herbrand, il est clair qu'une telle clause n'existera pas.
De plus, nous pouvons montrer que l'ensemble des formules atomiques qui sont des conséquences logiques d'un ensemble de clauses définies A est l'ensemble $\bigcap m(A)$ [KOWA 76] .

Théorème : Soit A un ensemble de clauses définies, alors $m(A) = \{c \in B_A \mid c \text{ est une conséquence logique de } A\}$.

démonstration : c est une conséquence logique de A
ssi $A \cup \{\neg c\}$ est non satisfaisable
ssi $A \cup \{\neg c\}$ n'a pas de modèle d'Herbrand

ssi ¬c est faux dans tous les modèles de de Herbrand de A

ssi c est vrai dans tous les modèles de de Herbrand de A

ssi $c \in \cap m(A)$

Les définitions énoncées c-dessus nous permettent de définir les conséquences logiques d'un ensemble de clauses. Cependant, les conséquences logiques ne seront que des informations positives. En effet, considérons A un ensemble de clauses et C une formule atomique. Nous ne pourrons pas prouver que ¬ C est une conséquence logique de A puisque $A \cup \{C\}$ admet B_A comme modèle et donc est satisfaisable. Cependant, il se pourrait que C ne soit pas non plus une conséquence logique de A et leur négation n'est pas non plus une conséquence logique de A. La définition sémantique de la règle d'inférence C W A consiste à considérer que la négation des faits de $B_A \setminus M_A$ est vraie, c.à.d. si C n'est pas une conséquence logique de A alors ¬C en est une.

2.1.2.2.2.2 Définition syntaxique de la C.W.A.

Dans le but de mettre en oeuvre la C,W.A, nous devons montrer que C n'est pas une conséquence logique de A. Dans ce cas, nous pouvons conclure que C est une conséquence logique de A. Cependant, nous savons que la logique du premier ordre est indécidable et qu'aucune procédure de preuve ne pourrait nous dire si C est une conséquence logique ou non de A. Par conséquent, l'application de C W A est généralement restreinte aux formules atomiques de B_A dont on peut montrer, en un temps fini, qu'elles ne sont pas des conséquences logiques de A. La règle d'inférence ainsi obtenue est appelée la négation par échec. La négation par échec a été largement étudiée dans [CLAR 78] [LLOY 84] . C'est cette règle d'inférence qui est mise en oeuvre dans Prolog et qui peut être reliée à l'opérateur de négation de Planner [Hewi 69]

Cependant, dans le cadre des bases de données déductives définies, nous nous limitons à des clauses définies sans symboles de fonction. Par conséquent, l'univers d'Herbrand et la base d'Herbrand sont des ensembles finis et il est possible de

déterminer si une formule atomique est une conséquence logique d'un ensemble de clauses. Par conséquent, il devrait être possible de construire un démonstrateur de théorèmes qui met en oeuvre la règle d'inférence C.W.A.
La définition sytaxique de C.W.A pour un ensemble de clauses A est $\{ \neg C \text{ est vrai} \mid A \nvdash C \}$
Dans le cas des clauses définies, comme nous pouvons déterminer si toute formule est une conséquence logique d'un ensemble de formule, la règle d'inférence C W A et la règle d'inférence de négation par échec sont équivalentes.

2.1.2.2.2.3 Cohérence des bases de données déductives définies sous C.W.A.

Etant donné une base de données déductives définies, les faits négatifs déduits à l'aide de la règle C W A peuvent être ajoutés aux faits positifs. Encore faut-il s'assurer que l'ensemble obtenu soit **cohérent.**
Soit un ensemble A de clauses.
Soit $\bar{A}$ l'ensemble des faits négatifs déduit de A à l'aide de C.W.A.
Nous dirons que A est cohérent avec C W A ou sous C W A si $A \cup \bar{A}$ est cohérent.
Reiter [Reit 78] a montré que si A est un ensemble de clauses définies alors A est cohérent sous C W A
Par conséquent, une base de données déductive est toujours cohérente sous C.W.A.

2.1.2.2.3 Définition opérationnelle d'une base de données déductive définie.

Une base de données déductive définie peut être définie par

(1) Une théorie T dont les axiomes propres sont
- les faits élémentaires
- les règles de déduction.

(2) Un ensemble IC de contraintes d'intégrité
(3) Une règle d'inférence appelée C.W.A (ou négation par échec).

La réponse aux questions, la vérification des contraintes d'intégrité se fait de façon identique, si ce n'est que le symbole "$\vdash$" doit être interprété comme "prouvable sous la C.W.A" et que les questions, les contraintes d'intégrité et les règles de déduction doivent être des formules à champ restreint.

Remarquons cependant que si les définitions, opérationnelle et en logique du premier ordre, sont équivalentes du point de vue de la réponse aux questions et de la vérification des contraintes d'intégrité, elles ne sont pas équivalentes du point de vue formel.
Pour montrer la distinction qu'il existe entre les deux définitions, il est nécessaire de définir le concept de logique monotone et non monotone.
Nous dirons qu'une logique est monotone si, étant donné une théorie T pour laquelle W est un théorème ($T \vdash W$), l'ajout d'un ensemble non vide d'axiomes à cette théorie permet toujours de déduire W. ($T \cup \{A_1, \ldots, A_n\} \vdash w$).
Si une logique n'est pas monotone, elle est non-monotone.

La logique des prédicats du premier ordre utilisée pour la définition formelle d'une base de données est monotone.
Cependant, la règle d'inférence C.W.A est une règle non monotone. En effet, considérons un ensemble de clauses $A = \{P \vee \neg Q\}$. Sous C W A nous pouvons conclure que $\neg P$ et $\neg Q$ sont des théorèmes ($A \vdash \neg P$ et $A \vdash \neg Q$). Supposons maintenant que nous ajoutions la clause "Q" à la base de données. Par conséquent, $\neg P$ et $\neg Q$ ne sont plus des théorèmes. ($A \cup \{Q\} \nvdash \neg Q$ et $A \cup \{Q\} \nvdash \neg P$). Par conséquent, nous sommes enprésence d'une logique non-monotone. Nous aurons l'occasion de revenir dans la troisième partie, sur les logiques non-monotones.

2.1.2.3 Conclusion.

Nous avons présenté deux définitions d'une base de données déductive définie : une définition en logique du premier ordre et une définition opérationnelle. La définition opérationnelle a ramené la théorie de la base de données déductives à un ensemble de clauses définies. Pour ce faire, nous avons supprimé l'axiome de fermeture du domaine en restreignant les questions, les contraintes et les règles de déductions à des formules à champ restreint. Ensuite, les autres axiomes ont été supprimés en considérant une nouvelle règle d'inférence C W A dont nous avons donné la définition sémantique et syntaxique. La théorie

de la base de données déductives étant un ensemble de clauses définies, une stratégie comme SLD-résolution pourra être mise en oeuvre. De même, la règle d'inférence C W A ou négation par échec pourra être implantée de manière efficace.

La définition opérationnelle d'une base de données déductive définie peut être exploitée de deux manières différentes. D'une part, nous pouvons augmenter une base de données conventionnelle de telle manière qu'elle prenne en compte les règles de déduction. On parle alors de bases de données déductives définies. D'autre part, il est possible d'augmenter un langage de programmation logique avec un composant "base de données" qui serait chargé de de la gestion des clauses à champ restreint sans symboles de fonction et des faits élémentaires. On parle alors de bases de données logiques. Dans ce qui suit, nous nous concentrerons essentiellement sur les bases de données déductives indéfinies qui ne font aucune restriction sur les formes des clauses si ce n'est bien sûr qu'elles soient sans symboles de fonction.

2.1.3 Les bases de données déductives indéfinies.

Ces bases de données déductives sont appelées "indéfinies" car les réponses aux questions ne sont pas toujours définies mais peuvent se présenter sous forme de disjonctions. En effet, si la seule information concernant le symbole de prédicats P est la clause $P(a) \vee P(b)$, la réponse à la question P (x) où x est une variable sera x = a+B ce qui signifie que P(a) est vrai ou que P(b) est vrai ou encore que $P(a) \wedge P(b)$ est vrai mais que le système ne dispose pas de suffisamment d'informations pour trancher entre ces alternatives. Le problème des réponses indéfinies est abordé dans [Reit 78].
Dans un premier temps, nous allons donner la définition d'une base de données déductive indéfinie. Ensuite, nous montrerons qu'une base de données déductive indéfinie n'est pas cohérente sous C.W.A. Cette hypothèse sera ensuite généralisée pour donner lieu à G.C.W.A [Mink 82]. Nous terminerons par une brève discussion sur la prise en compte de valeurs nulles dans les bases de données déductives indéfinies.

2.1.3.1 Définition d'une base de données déductive indéfinie.

Une base de données déductive indéfinie peut être définie par

(1) une théorie dont les axiomes propres sont

- un ensemble de faits élémentaires de la forme

$P \leftarrow$ où P est une formule atomique de base

- un ensemble de règles de déduction de la forme

$$P_1 \vee \ldots \vee P_n \leftarrow Q_1 \wedge \ldots \wedge Q_m \qquad n \geqslant 0 \qquad m \geqslant 0.$$

(2) une règle d'inférence GCWA (generalised closed world assumption)

(3) un ensemble IC de contraintes d'intégrité qui sont des formules bien formées fermées.

Les règles de déduction, les questions et les contraintes d'intégrité sont des formules à champ restreint.

2.1.3.2 L'hypothèse du monde fermé généralisée.

2.1.3.2.1 Incohérence d'une base de données déductive indéfinie sous C.W.A.

Considérons une base de données déductive indéfinie qui contient les clauses $\{P(a) \vee P(b), P(c) \vee \neg P(d)\}$.

La règle d'inférence C W A nous permet de déduire

$$\{\neg P(a), \neg P(b), \neg P(d), \neg P(c)\}$$

puisque P(a),P(b),P(c),P(d) ne sont pas des conséquences logiques de la base de données. Par conséquent, nous obtenons l'ensemble

$$\{P(a) \vee P(b), P(c) \vee \neg P(d), \neg P(a), \neg P(b), \neg P(d), \neg P(c)\}.$$

en ajoutant les informations négatives à la base de données. Manifestement, cet ensemble est incohérent de sorte que nous ne pourrons pas appliquer la règle d'inférence C.W.A pour les bases de données déductives indéfinies. Remarquons cependant que l'ajout de $\neg P(c)$ et $\neg P(d)$ n'aurait pas donné lieu à un ensemble incohérent ce qui nous amènerait à croire que dans certains cas, le fait pour une formule atomique de ne pas être une conséquence logique nous permet de supposer que sa négation est vraie. Ces réflexions ont donné lieu à une généralisation de la règle C.W.A.

2.1.3.2.2 Définition sémantique de G.C.W.A.

Pour donner une définition sémantique de G.C.W.A, nous devrons à nouveau nous reporter aux interprétations et modèles de Herbrand. Nous avions défini M(DB) comme l'ensemble des modèles d'un ensemble de clauses DB. Lorsque les clauses se réduisaient à des clauses de Horn, nous avions vu que l'intersection de tous les modèles était encore un modèle.

Cette propriété n'est plus vraie si nous sortons du cadre des clauses de Horn.

En effet, considérons la clause $P(a) \vee P(b)$.
Cette clause admet deux modèles à savoir :

$$M_1 = \{P(a)\} \qquad M_2 = \{P(b)\}$$

Il est clair que l'intersection de ces deux modèles est vide et donc ne sera pas modèle de cette clause. Nous définissons alors la notion de modèle minimal d'un ensemble de clauses.
Une interprétation de Herbrand I est un modèle minimal de DB ssi

1) $I \in M(DB)$
2) il n'existe pas de modèle $I' \in M(DB)$ qui sont un sous-ensemble de I.

L'ensemble des modèles minimaux de DB est noté MM(DB).

Exemple : Soit $DB = \{P(a) \vee P(b), P(c) \vee \neg P(d)\}$,
l'ensemble des modèles de Herbrand de DB, M(DB) est :

$$\left\{ \begin{array}{l} \{P(a)\} \\ \{P(b)\} \\ \{P(a), P(b)\} \\ \{P(a), P(c)\} \\ \{P(b), P(c)\} \\ \{P(a), P(b), P(c)\} \\ \{P(a), P(c), P(d)\} \\ \{P(b), P(c), P(d)\} \\ \{P(a), P(b), P(c), P(d)\} \end{array} \right\}$$

les modèles minimaux de DB sont :

$$\{P(a)\} \text{ et } \{P(b)\}.$$

Nous avons vu précédemment que l'ajout de $\neg P(c)$ et $\neg P(d)$ rendait un ensemble cohérent. Remarquons ici que P(c) et P(d) ne figurent dans aucun modèle minimal, ce qui n'est manifestement pas le cas de P(a) et P(b).
De même, nous pouvons remarquer que l'intersection de tous les modèles est l'ensemble vide $\emptyset$ qui est une interprétation de Hrebrand mais qui n'est pas un modèle.

Considérons donc $MM(DB) = \{m_i\}$ $(1 \leq i \leq m)$ $m > 0$ l'ensemble des modèles minimaux d'un ensemble cohérent de clauses.
Soit $\mathcal{M} = \{\alpha \mid \alpha \in \bigcap_{i=1}^{m} m_i\}$

Soit $\overline{\mathcal{M}} = \{\alpha \mid \alpha \in \bigcap_{i=1}^{m} (B_{DB} - m_i)$ où B_{DB} est la base de Herbrand de DB.

$\mathcal{M}$ représente l'ensemble des éléments de B_{DB} qui sont dans tous les

modèles minimaux tandis que $\overline{\mathcal{M}}$ représente l'ensemble des éléments de B_{DB} qui ne sont dans aucun modèle minimal.
Soit $\mathcal{M}_D = \mathcal{M} \cup \overline{\mathcal{M}}$ l'ensemble des formules définies.
Soit $\mathcal{M}_I = B_{DB} \setminus \mathcal{M}_D$ l'ensemble des formules indéfinies.
L'ensemble des formules définies est l'ensemble des formules pour lesquelles on peut décider si elles sont vraies ou si leurs négations sont vraies.
Les formules dans $\mathcal{M}$ sont les formules vraies. Les formules dans sont les formules dont la négation peut être considérée comme vraie. Ce sont ces dernières qui constituent la G.C.W.A.

Exemple 1 : $DB = \{ P(a) \vee P(b) \}$ $M_1 = \{ P(a) \}$ $M_2 = \{ P(b) \}$ où m_i est un modèle minimal $(1 \leq i \leq 2)$

$$\mathcal{M} = \varnothing \quad \overline{\mathcal{M}} = \varnothing \quad \mathcal{M}_I = \{ P(a), P(b) \}$$

Exemple 2 : $DB = \{ Q(a), P(a) \vee P(b) \}$
$M_1 = \{ Q(a), P(a) \}$
$M_2 = \{ Q(a), P(b) \}$

$$\mathcal{M} = \{ Q(a) \} \quad \overline{\mathcal{M}} = \{ Q(b) \} \quad \mathcal{M}_I = \{ P(a), P(b) \}.$$

Etant donné deux formules atomiques appartenant à $\mathcal{M}_I$ par exemple R(a) et t(b) on peut se demander si $R(a) \vee t(b)$ est vrai. Pour ce faire, il est nécessaire de regarder si dans tout modèle de l'ensemble de clauses considéré R(a) ou t(b) ou encore les deux, sont vrais.

Enfin, nous remarquerons que G.C.W.A est une généralisation adéquate de C.W.A. En effet, si nous considérons un ensemble de clauses de Horn, il existe un seul modèle minimal qui est l'intersection de tous les modèles. Par conséquent, $\overline{\mathcal{M}}$ contient toutes les formules qui n'appartiennent pas à l'intersection, ce qui correspond à la définition de C.W.A.

2.1.3.2.3 Définition syntaxique de G.C.W.A.

Considérons DB un ensemble cohérent de clauses.
Soit $E = \{ P \mid DB \vdash P \vee Q$ où P est une formule atomique de base et où Q est soit la clause vide, soit une disjonction de formules atomiques . E est donc l'ensemble des formules atomiques qui ap-

partiennent à au moins une disjonction de formules atomiques dérivables de DB.
Dès lors $\overline{DB} = B_{DB} \setminus \in$ est l'ensemble de toutes les formules dont la négation peut être supposées vraies dans DB . $\overline{DB}$ correspond à la définition syntaxique de G.C.W.A.

Le problème de cette définition syntaxique est que, contrairement à la définition de C.W.A qui pouvait donner lieu à une mise en oeuvre naturelle et efficace, elle sera pratiquement de peu d'utilité et que des concepts plus pratiques devront être développés pour une mise en oeuvre efficace des bases de données déductives indéfinies

2.1.3.3 Traitement des valeurs nulles dans les bases de données déductives indéfinies.

Gallaire, Minker, Nicolas [GALL 84] mentionnent que la prise en compte de l'hypothèse généralisée du monde fermé permet de traiter un certain type de valeurs nulles. Il s'agit de valeurs nulles qui représentent une valeur inconnue qui se trouve parmi les constantes de la base de données. En effet, considérons la base de données qui consiste en un ensemble de clauses.

$\{ P(w), Q(a), Q(b) \}$ où w est une valeur nulle ou une constante de Skolem due à une information du type $\exists x\ P(x)$.
Si les constantes de la base de données sont a et b alors l'information $\exists x\ P(x)$ sachant que x est dans les constantes de la base de données peut être remplacée par P(a) P(b).
C'est pourquoi l'information contenue dans **P(w)** est identique à celle contenue dans $P(a) \vee P(b)$. La base de données où nous avons remplacé **P(w)** par $P(a) \vee P(b)$ ne permettra pas de conclure $\neg P(a)$ sous G C W A et par conséquent la valeur nulle est traitée correctement.

Nous remarquerons cependant que, si théoriquement ce résultat est très beau, il est assez restrictif de considérer que la valeur nulle doit être parmi les constantes de la base de données. De plus, ce traitement de la valeur nulle nous obligera à mémoriser des listes de disjonctions qui peuvent être extrèmement longues dans la base de données. Pratiquement donc, cette solution parait peu réaliste.

2.1.3.4 Conclusion.

Nous avons donc présenté les bases de données déductives indéfinies.

Les règles de déduction ne se limitent plus cette fois à des clauses définies. Cependant, nous avons remarqué que ces bases de données n'étaient plus cohérentes sous C.W.A et nous avons introduit G.C.W.A. Nous avons remarqué que si G.C.W.A est une notion bien définie sémantiquement, sa mise en oeuvre effective présente des problèmes Nous remarquerons également que si les bases de données déductives définies pouvaient être ramenées à un ensemble de clauses définies pour la définition de leur théorie, par définition les bases de données déductives indéfinies ne font pas de restriction sur les clauses et demanderont donc une stratégie de résolution complète pour des clauses quelconques comme, par exemple, model elimination [LOVE 69] SL-résolution [KOWA 71].
Enfin, nous avons mentionné que les bases de données déductives indéfinies pouvaient traiter un certain type de valeurs nulles mais qu'encore une fois, une solution pratique devrait être élaborée.
Le problème des informations incomplètes sera traité dans la troisième partie dans le cadre des logiques modales.

2.1.4 Séparation des lois générales en règles de déduction et contraintes d'intégrité.

Dans ce qui suit, nous nous placerons dans le cadre des bases de données déductives définies. Remarquons d'emblée qu'il n'existe pas de critère absolu pour nous dire si une loi générale doit être utilisée comme contrainte d'intégrité ou comme règle de déduction. Cette décision est du ressort de la conception d'un schéma de base de données. Par ailleurs, nous remarquerons que si les règles de déduction, les questions et les contraintes sont exprimées sous forme de wff, elles ne contiendront pas de symboles de fonction. Cette contrainte est imposée pour avoir des réponses explicites et définies aux questions soumises à la base de données. Nous distinguerons essentiellement deux critères qui permettront d'orienter le choix d'utilisation d'une règle générale : les critères syntaxiques et les critères sémantiques.

2.1.4.1 Les critères syntaxiques.

1. Considérons une loi générale exprimée sous forme d'une wff quelconque. Si sa transformation sous forme de clauses fait apparaitre soit un symbole de fonction, soit une constante de Skolem alors elle

devra être utilisée comme contrainte d'intégrité. Nous avons déjà mentionné plus haut que les symboles de fonction devaient être exclus des règles de déduction pour avoir des réponses explicites et définies aux questions. En conséquence, il sera impossible dans une base de données déductives de traiter comme règle de déduction des informations de la forme $\forall x \exists y$ père (y,x) ou $\exists x$ P(a,x).
Dans le premier cas, la transformation sous forme de clause fait apparaitre un symbole de fonction, par exemple, père(f(x),x).
Dans le second cas, on fait apparaitre une constante de Skolem, P(a,w) où w est la constante de Skolem introduite.

2. Nous avons énoncé qu'une base de données déductive définie n'admettait que des clauses définies comme règles de déduction.
Dès lors, toute loi générale exprimée sous forme de clauses qui ne serait pas une clause définie, devrait être utilisée comme contrainte d'intégrité.

3. Si, dans une clause définie de la forme

$$A \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_n$$

A est une relation qui n'est pas définie dans la base de données par l'utilisateur alors cette clause doit être utilisée comme contrainte d'intégrité. Utilisée en tant que règle de déduction, elle ne générerait que des faits faux ou incohérents. Nous pensons particulièrement aux relations de comparaison.

exemple 1 : L'information générale
'Si x est une personne et si y est l'âge de x alors y est inférieur à 150" est traduite sous forme de clause de Horn par $(y < 150) \leftarrow$ âge $(x,y) \wedge$ personne (x).
Cette règle ne produira jamais, si elle est utilisée comme règle de déduction
"10 < 150" ou des faits incohérents de la forme "170 < 150".
Manifestement, elle doit être utilisée comme contrainte d'intégrité.

exemple 2 : L'information générale
"Toute personne n'a qu'un père"
traduite sous forme de clause de Horn
$(y = y') \leftarrow$ père$(x,y) \wedge$ père(x,y')
est également une règle qui doit être utilisée comme contrainte d'intégrité.

2.1.4.2 Critères sémantiques : quelques réflexions.

1. Le dernier critère syntaxique que nous venons d'énoncer peut être généralisé. En effet, si une loi générale

$$A \leftarrow P_1 \wedge P_2 \wedge \ldots \wedge P_n$$

n'apporte rien à la définition de A, elle devrait être utilisée comme contrainte d'intégrigé. La relation A peut être entièrement définie par d'autres règles ou faits et l'utilisation de cette loi en tant que règle de déduction ne génèrerait, à nouveau, que des faits incohérents ou redondants. Le but de cette règle est plus que probablement de réduire les états valides de la base de données.
Considérons, par exemple, l'information générale

"Peter n'aime que les livres"

Cette information peut être représentée sous forme de clause de **Horn** par **livre(x)** ← **aime(Peter,x)** où x est une variable.
Cette clause ne peut pas être considérée comme tout ou partie de la définition de la relation " **livre(x)**" qui sera plus que probablement définie, de manière bien plus adéquate, ailleurs dans la base des données Plutôt, elle exprime une contrainte d'intégrité qui permet d'interdire l'insertion d'un fait " **aime**(Peter,y)" où y est une constante qui n'est pas un livre.

2. L'utilisation d'une loi générale, en tant que règle de déduction, est une manière d'assurer que la contrainte d'intégrité, qui peut être également exprimée par cette loi, soit toujours et automatiquement vérifiée, évitant ainsi des incohérences et des insertions qui pourraient être effectuées automatiquement.
Considérons, la loi générale

$$\text{grand-père}(x,y) \leftarrow \text{père}(x,z) \wedge \text{père}(z,y) \quad (1)$$

cette loi sera plus adéquate si elle est utilisée comme règle de déduction que comme contrainte d'intégrité. Considérons la base de données dans l'état

$$\{\ \text{père}(a,b)\ \}$$

Si la loi générale (1) est utilisée comme contrainte d'intégrité, l'insertion du fait " père(b,c)" a pour effet d'invalider la contrainte exprimée par (1). Cette insertion sera rejetée et nous devrons insérer "grand-père(a,c) " avant d'insérer " père(b,c)". La même loi, utiliée comme règle de déduction, aurait permis l'insertion de " père(b,c)".
De plus, le fait "grand-père(a,c) " est également vrai dans cette base

de données puisqu'il peut être déduit des faits élémentaires et de la règle de déduction. Par conséquent, l'utilisation de cette loi en règle de déduction nous assure que la contrainte d'intégrité exprimée par (1) est aussi vérifiée dans la base de données. L'utilisation de cette loi en tant que règle de déduction est donc plus adéquate et, de plus, elle libèrera l'utilisateur de l'insertion des faits dans la relation "grand-père".

La détermination des critères sémantiques permettant de choisir entre l'utilisation d'une loi générale comme règle de déduction ou comme contrainte d'intégrité reste un sujet de recherche.

2.2 Mise en oeuvre des règles de déduction.

2.2.0 Introduction.

La théorie d'une base de données déductive définie est constituée de faits élémentaires et de règles de déduction. Afin de répondre aux questions que nous pourrions poser à cette base de données, il conviendra, à un moment où à un autre, de mettre en oeuvre ou d'exploiter la connaissance des règles de déduction. Pour ce faire, deux grandes approches s'offrent à nous : l'approche en génération et l'approche en dérivation.

L'approche en génération exploite les règles de déduction lors de l'insertion ou de la suppression des faits élémentaires ou des règles. La base de données contiendra, outre les faits élémentaires et les règles de déduction, tous les faits qui sont des conséquences logiques des faits élémentaires et des règles de déduction. L'interrogation de la base de données ne réclamera aucune déduction et sera semblable à celle d'une base de données conventionnelle. De plus, et cela ne sera pas vrai pour l'approche en dérivation, le processus de génération de faits se terminera toujours sans qu'aucune condition d'arrêt particulière ne soit nécessaire. Par contre, étant donné qu'il est possible de supprimer des faits élémentaires ou des règles de déduction, il sera nécessaire d'assurer une maintenance des informations déduites.

L'approche en dérivation consiste à exploiter les règles de déduction lors de la détermination de la ou des réponses à une question. La base de données ne contiendra que les faits élémentaires et les règles de déduction. Le mécanisme d'évaluation des questions sera donc modifié pour prendre en compte les règles de déduction. Pour ce faire, deux méthodes peuvent être utilisées, la méthode compilée et la méthode interprétée, selon que les processus de déduction et d'interaction avec la base de données soient séparés ou non. Cependant, quelle que soit la méthode choisie, les règles récursives poseront des problèmes.

Dans ce qui suit, nous considérerons respectivement l'approche en génération et l'approche en dérivation. Dans l'approche en dérivation, nous présenterons la méthode interprétée et compilée.

2.2.1 L'approche en génération.

2.2.1.0 Introduction.

Dans une base de données déductive mise en oeuvre selon l'approche en génération, le mécanisme de déduction est activé lors de l'insertion et la suppression de faits élémentaires et de règles de déduction. L'approche en génération peut être comparée aux théorèmes "antécédent" et "erase" du langage PLANNER [HEWI 69].

Dans ce qui suit, nous examinerons comment on peut mettre en oeuvre une base de données déductive selon l'approche en génération. Après un certain nombre de définitions, nous donnerons une spécification des deux opérations principales d'une base de données déductives : l'insertion d'un fait et la suppression d'un fait.
Nous exposerons la stratégie d'insertion pour ensuite définir et démontrer un certain nombre de propriétés.
Nous en déduirons l'algorithme et tenterons d'en démontrer la correction et la terminaison. Nous parlerons ensuite du problème de la complexité de cet algorithme et terminerons par une présentation de la suppression.

2.2.1.1 Objectif.

Les informations contenues dans une base de données déductive (BDD) peuvent généralement être classées en trois catégories :

- les informations explicites, c.à d. un tuple d'une des relations définies dans la BDD qui a été explicitement (par un utilisateur) caractérisée comme appartenant à la BDD.
- les règles de déduction qui caractérisent les informations que l'on peut considérer comme appartenant à la BDD, même si celles-ci ne sont pas explicites.
- les informations implicites, c.à d. un tuple d'une des relations définies dans la BDD qui peut être déduite compte tenu des autres informations et des règles de déduction.

Une base de données saturée (BDDS) est une BDD dans laquelle toutes les informations implicites sont présentes.
Dans cette partie, nous nous efforcerons de traiter le problème de l'insertion et de la suppression d'une information dans une BDDS. Cela revient à déterminer, à partir d'une BDDS et d'une information inf, une BDDS dont les informations explicites sont celles de la BDDS donnée auxquelles on a ajouté (resp. supprimé) l'information inf et dont les règles de déduction sont identiques à celles de la BDDS donnée.

Pour aborder ce problème, nous commencerons par définir quelques concepts et par donner une spécification plus technique. Nous construirons ensuite un algorithme et tenterons d'en démontrer la validité ainsi que la terminaison. Nous examinerons également la complexité théorique de cet algorithme. Cette présentation sera accompagnée d'une comparaison des stratègies choisies lors de la construction des algorithmes avec d'autres alternatives possibles.

2.2.1.2 Définitions.

Dans ce qui suit, nous allons définir un certain nombre de concepts relatifs à une BDD qui nous seront utiles pour la spécification du problème.

Nous noterons $\mathscr{C}$, l'ensemble fini des constantes qui est l'union des différents domaines de la BDD.

Nous noterons $\mathscr{P}$, l'ensemble fini des symboles de prédicat représentant l'ensemble des noms de relations définies dans la BDD.

Nous appellerons fait, une formule atomique $P(t_1,\ldots,t_n)$

où $n \geqslant 1$, $P \in \mathcal{P}$ et $t_i \in \mathcal{C}$

Nous appellerons faits explicites de la BDD, l'ensemble des faits explicitement présents dans la BDD.

Nous appellerons règle de déduction, une clause de Horn de la forme

$$A \leftarrow B_1 \wedge \ldots \wedge B_k$$

où $k \geqslant 1$ et A, B_i sont des formules atomiques de la forme $P(v_1, \ldots, v_n)$ où $n \geqslant 1$, $P \in \mathcal{P}$ et v_i est une variable ou $v_i \in \mathcal{P}$.

Nous appellerons faits explicites de la BDD, l'ensemble des faits explicitement présents dans la BDD.

Soit FE, les faits explicites de la BDD.

Soit RD, les règles de déduction de la BDD.

Nous appellerons faits implicites associés à FE et RD, l'ensemble des faits qui sont conséquences logiques de la théorie FE ∪ RD.

A toute BDD est associée une BDDS formée des trois composants $\langle FE, RD, FI \rangle$ FE étant les faits explicites de BDD, RD les règles de déduction et FI les faits implicites associés à FE et RD. Remarquons que, par définition de FI, $FE \subset FI$.

2.2.1.3 Spécification.

Soient $\mathcal{F}$: l'ensemble des faits.

$\mathcal{B}$: l'ensemble des BDDS pour un $\mathcal{C}$ et un $\mathcal{P}$ donnés.

2.2.1.3.1 Insert(BDDS,f)

Soient BDDS $\in \mathcal{B}$ avec BDDS = $\langle FE, RD, FI \rangle$

$f \in \mathcal{F}$

Si $f \in FE$ alors le résultat de cette opération est BDDS.

Si $f \in FI \setminus FE$ alors le résultat de cette opération est $\langle FE \cup \{f\}, RD, FI \rangle$.

Sinon le résultat de cette opération est $\langle FE \cup \{f\}, RD, FI' \rangle$ où FI' est l'ensemble des faits implicites associés à $FE \cup \{f\}$ et RD.

Insert est donc une opération de la forme

$$\text{Insert} : \mathcal{B} \times \mathcal{F} \rightarrow \mathcal{B}$$

2.2.1.3.2 Suppress(BDDS,f)

Soient BDDS $\in \mathcal{B}$ avec BDDs = $\langle FE, RD, FI \rangle$.

$f \in \mathcal{F}$.

Si f ∈ FE alors le résultat de cette opération est $\langle FE \setminus \{f\}, RD, FI' \rangle$
où FI' sont les faits implicites associés à $FE \setminus \{f\}$ et RD.
Sinon l'opération n'est pas définie.

Suppress est donc une opération de la forme

$$\text{Suppress} : \mathcal{B} \times \mathcal{F} \rightarrow \mathcal{B}$$

2.2.1.3.3 Justification.

Dans ce paragraphe, nous tenterons de montrer que la spécification de ces deux opérations rencontre bien les notions d'insertion et de suppression de faits dans une base de données déductive telles qu'elles ont été présentées dans la section 1.

Insertion d'un nouveau fait explicite.

Si le fait à insérer est un fait explicite ($f \in FE$) alors
la BDDS résultat est identique à la base de données saturée initiale puisque l'ensemble des faits explicites est inchangé; de même, aucune nouvelle information implicite ne pourra être déduite puisque l'ensemble des faits explicites et l'ensemble des règles de déduction sont inchangés.

Si le fait à insérer est un fait implicite mais non explicite ($f \in FI \setminus FE$) alors dans ce cas, l'ensemble des faits explicites de la BDDS résultat est la réunion des faits explicites de la BDDS initiale et du nouveau fait explicite ($FE \cup \{f\}$). Par contre, l'ensemble des faits implicites de la BDDS résultat est identique à celui de la BDDS initiale puisque la logique des prédicats du premier ordre [EEBI 84] nous dit que :

Si CL est l'ensemble des conséquences logiques de $FE \cup RD$
$f \in CL$
alors CL est l'ensemble des conséquences logiques de
$FE \cup RD \cup \{f\}$.

Si le fait à insérer n'est pas un fait implicite (et donc pas un fait explicite) ($f \notin FI$) alors, comme dans le cas précédent, l'ensemble des faits explicites de la BDDS résultat est la réunion des faits explicites de la BDDS initiale et du nouveau fait explicite ($FE \cup \{f\}$). Comme f n'est pas une conséquence logique $FE \cup RD$, l'ensemble des conséquences logiques de $FE \cup \{f\} \cup RD$ est plus vaste. Nous aurons donc, pour la BDDS résultat, un nouvel ensemble de faits implicites associés à $FE \cup \{f\}$ et RD.

Suppression d'un fait explicite.

Nous ne prendrons en compte que la suppression de faits explicites d'une BDD car pour que la suppression d'un fait implicite ait un sens, il faudrait que celui-ci ne fasse plus partie des conséquences logiques des faits explicites et des règles de déduction. Cela demanderait donc de supprimer également un sous-ensemble de faits explicites (ou de règles de déduction). Lorsque le fait à supprimer est un fait explicite, les faits explicites de la BDDS résultat sont ceux de la BDDS initiale auxquels on a retiré ce fait explicite. L'ensemble des conséquences logiques de $(FE \setminus \{f\}) \cup RD$ pourrait être moins vaste que celui de $FE \cup RD$ et, par conséquent, la BDDS résultat aura un nouvel ensemble de faits implicites associé à $(FE \setminus \{f\})$ et RD.

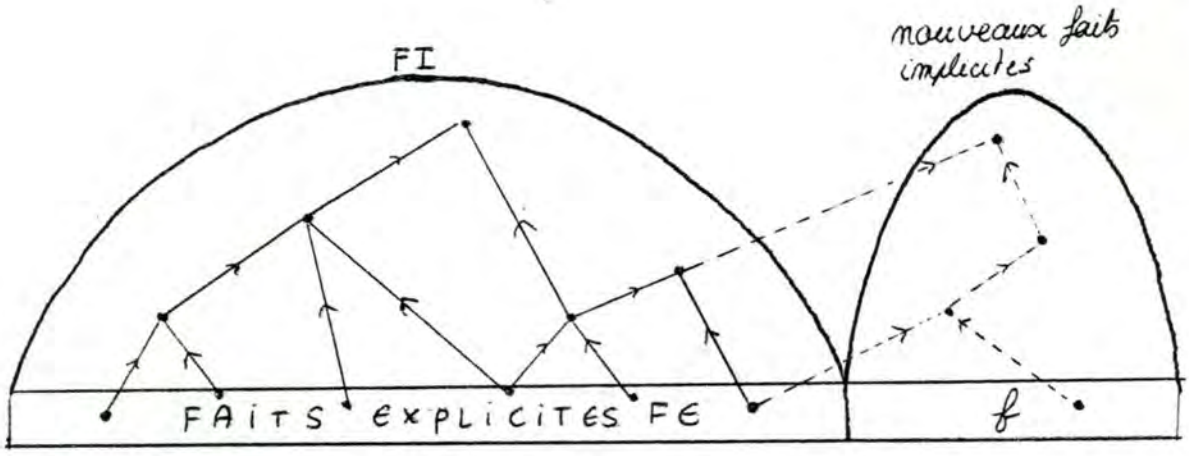
2.2.1.3.4 Convention.

Au lieu de passer BDDS comme paramètre, nous supposerons l'existence d'une BDDS globale constituée des 3 variables globales FE, RD et FI contenant une représentation de BDDS. Aux opérations Insert(BDDS, f) et Suppress(BDDS, f) correspondront donc deux programmes effectuant respectivement les affectations

BDDS ← Insert(BDDS, f)

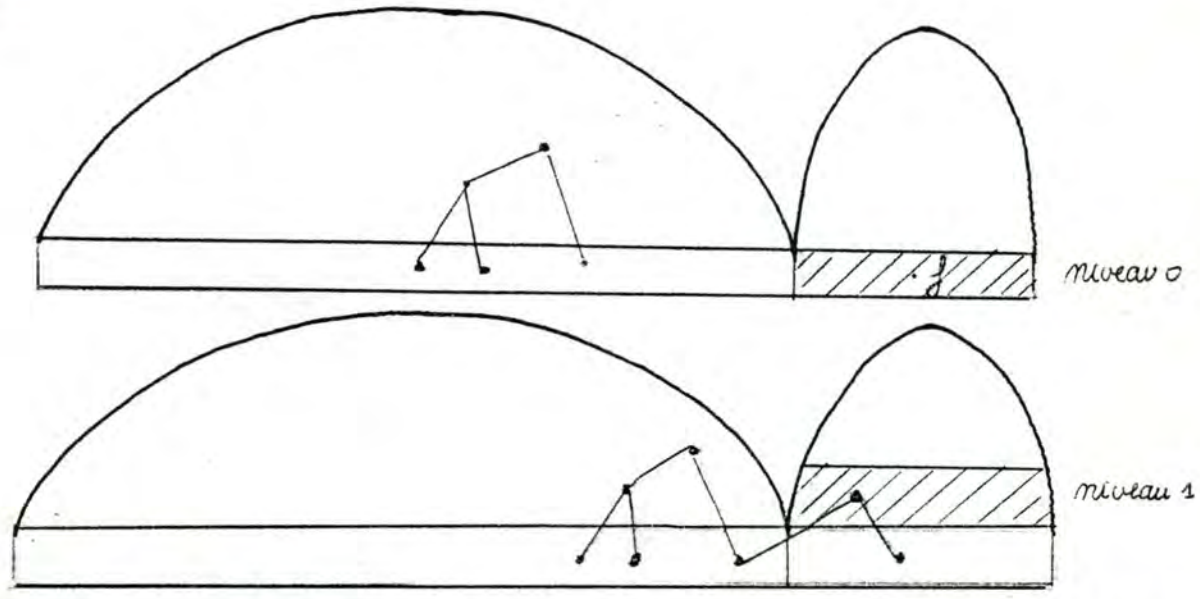
BDDS ← Suppress(BDDS, f)

2.2.1.4 Principe de la stratégie utilisée pour l'insertion.

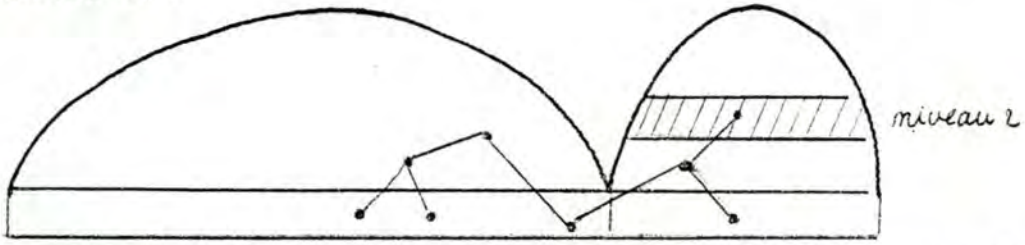


La stratégie envisagée consiste à parcourir (plus exactement à générer) l'ensemble des nouvelles conséquences logiques selon une succession de niveaux, chaque niveau correspondant aux nouveaux faits implicites déductibles à partir de FI et des nouveaux faits implicites générés aux niveaux inférieurs.

exemple.



Nous dirons qu'un fait est directement déductible s'il est obtenu en appliquant une seule fois une des règles de déduction. Le niveau 1 contient donc tous les nouveaux faits directement déductibles à partir de FI et de f (niveau 0).



Le niveau 2 contient donc tous les nouveaux faits directement déductibles à partir de FI et des nouveaux faits des niveaux 0 et 1.
Tous les nouveaux faits implicites seront déduits lorsqu'à un niveau i, celui-ci ne possèdera plus de nouveaux faits. L'existence d'un tel niveau est garanti par le fait que l'ensemble des faits est fini et que l'on génère un graphe sans circuit. On peut donc conclure qu'il existera un niveau sans descendant direct.

Remarque : analogie avec le parcours d'un graphe.

Le problème peut être vu comme le parcours des différents niveaux d'un graphe si l'on considère que

- les noeuds de niveau 0 sont les faits implicites, les faits explicites et le fait f.
- aucun sommet de niveau i ne possède d'ascendant de niveau $\geq$ i.

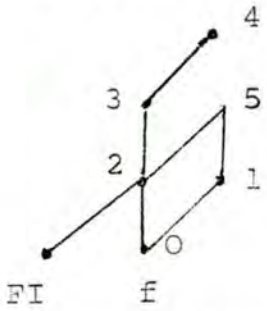
Remarquons enfin que le parcours des différents niveaux de ce graphe ne constitue ni un parcours en largeur d'abord ni un parcours en profondeur d'abord.

Autres stratégies possibles.

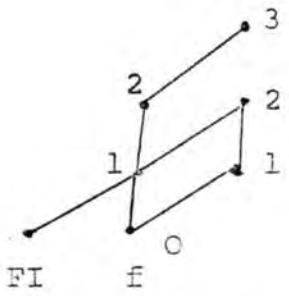
Bien que la stratégie énoncée ci-dessus nous paraisse la plus simple, signalons une autre stratégie dont le principe peut être schématisé de la manière suivante.

Tant que l'on peut déterminer un nouveau fait implicite f_1 directement déductible à partir de f,

réappliquer le processus pour f_1.



stratégie alternative



stratégie en niveaux

Nous préférons la stratégie en niveaux car elle permet de caractériser de manière plus aisée les faits implicites déjà générés à une étape quelconque de l'algorithme (Invariant sur l'état de la BDD). La seconde stratégie, quant à elle, ne permet pas une expression aussi aisée de l'état intermédiaire pour la simple raison qu'elle dépend de l'ordre de génération de chaque nouveau fait implicite.

2.2.1.5 Concepts liés à la stratégie.

Nous noterons $D^n_{RD}(F)$ l'ensemble des faits déductibles à n niveaux à partir d'un ensemble de faits F et d'un ensemble de règles de déduction RD. $D^n_{RD}(F)$ peut être défini de la manière suivante :

1) $D^o_{RD}(F) = F$

2) $D^n_{RD}(F) = \{ f \in \mathcal{F}$: il existe une règle "$A \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_k$" $\in$ RD et une substitution θ telles que

(1) $\forall\ 1 \leq i \leq k \quad B_i\theta \in D^m_{RD}(F)$ avec $m < n$

(2) $\exists\ 1 \leq j \leq k \quad B_j\theta \in D^{m-1}_{RD}(F)$

(3) $f = A\theta$ $\}$

Dans le cas où n = 1, les conditions (1) à (3) deviennent

(1)' $\forall\ 1 \leq i \leq k \quad B_i\theta \in F$

(2)' $f = A\theta$

Nous définirons également l'ensemble des faits déductibles jusqu'à n niveaux à partir d'un ensemble de faits F et d'un ensemble de règles de déduction RD, $\mathfrak{D}^n_{RD}(F)$ de la manière suivante.

1) $\mathcal{D}^{o}_{RD}(F) = F$

2) $\mathcal{D}^{n}_{RD}(F) = \bigcup_{i=o}^{n} D^{i}_{RD}(F)$

Propriété 1 Cette propriété exprime une caractéristique des nouveaux faits déductibles en n+1 niveaux

$D^{n+1}_{RD}(F) \setminus \mathcal{D}^{n}_{RD}(F) = D^{1}_{RD}(\mathcal{D}^{n}_{RD}(F)) \setminus \mathcal{D}^{n}_{RD}(F)$

démonstration :

$D^{n+1}_{RD}(F) \setminus \mathcal{D}^{n}_{RD}(F)$ est l'ensemble défini de la manière suivante

$\{ f \in \mathcal{F} \mid$ Il existe une règle "$A \leftarrow B_1 \wedge \ldots \wedge B_k$" $\in RD$ et une substitution θ telle que

(1) $\forall 1 \leqslant i \leqslant k$: $B_i\theta \in D^{m}_{RD}(F)$ avec $m < n+1$

(2) $\exists 1 \leqslant J \leqslant k$: $B_j\theta \in D^{n}_{RD}(F)$

(3) $f = A\theta$

(4) $f \notin \mathcal{D}^{n}_{RD}(F)$ $\}$

$D^{1}_{RD}(\mathcal{D}^{n}_{RD}(F)) \setminus \mathcal{D}^{n}_{RD}(F)$ est l'ensemble défini de la manière suivante

$\{ f \in \mathcal{F} :$ Il existe une règle "$A \leftarrow B_1 \wedge \ldots \wedge B_k$" $\in RD$ et une substitution θ telle que

(1') $\forall 1 \leqslant i \leqslant k$ $B_i\theta \in \mathcal{D}^{n}_{RD}(F)$

(3') $f = A\theta$

(4') $f \notin \mathcal{D}^{n}_{RD}(F)$ $\}$

Montrons que ces deux ensembles sont équivalents

(1) La condition (2) est redondante. En effet, si $f \in \mathcal{D}^{n}_{RD}(F)$ et que (1) et (3) sont vrais alors (2) sera faux

$f \in \mathcal{D}^{n}_{RD}(F)$ et (1) implique que

$\forall 1 < i \leqslant k$ $B_i\theta \notin D^{m}_{RD}(F)$ avec $m < n$

Considérons m' le plus grand entier tel qu'il existe $1 \leqslant i \leqslant k$ $B_i\theta \in D^{m}_{RD}(F)$.

On aura $m' < n$. Dès lors, si (3) est vrai alors

$f \in D^{m'+1}_{RD}(F)$ ou encore $f \in \mathcal{D}^{n}_{RD}(F)$ comme $m' < n$

Par contre, si $f \notin \mathcal{D}^{n}_{RD}(F)$ et que (1) et (3) sont vrais alors, par définition de $D^{n}_{RD}(F)$, (2) sera vrai

(2)La condition (1) et (1') sont équivalentes
car $B_i\theta \in D^m_{RD}(F)$ pour $m < n+1$
est équivalent à

$$B_i\theta \in \bigcup_{i=o}^{n} D^n_{RD}(F) = \mathfrak{D}^n_{RD}(F)$$

(3)Les conditions (3) et (4) sont trivialement équivalentes à (3') à (4')

Propriété 2 Cette propriété exprime une caractéristique des faits déductibles jusqu'à n+1 niveaux en fonction des faits déductibles jusqu'à n niveaux

$$\mathfrak{D}^{n+1}_{RD}(F) = \mathfrak{D}^n_{RD}(F) \cup D^1_{RD}(\mathfrak{D}^n_{RD}(F))$$

démonstration :

$$\mathfrak{D}^{n+1}_{RD}(F) = \mathfrak{D}^n_{RD}(F) \cup D^{n+1}_{RD}(F) \quad \text{définition de } {}^n_{RD}(F)$$

$$= \mathfrak{D}^n_{RD}(F) \cup (D^{n+1}_{RD}(F) \setminus \mathfrak{D}^n_{RD}(F))$$

Propriété 1

$$= \mathfrak{D}^n_{RD}(F) \cup (D^1_{RD}(\mathfrak{D}^n_{RD}(F)) \setminus \mathfrak{D}^n_{RD}(F))$$

$$= \mathfrak{D}^n_{RD}(F) \cup D^1_{RD}(\mathfrak{D}^n_{RD}(F))$$

Propriété 3 Si $D^1_{RD}(\mathfrak{D}^n_{RD}(F)) \setminus \mathfrak{D}^n_{RD}(F) = \emptyset$ alors

$$\forall m > n \quad D^m_{RD}(F) \subset \mathfrak{D}^n_{RD}(F)$$

démonstration :

Supposons que la propriété soit vraie pour $n < k \leq m$ et démontrons qu'elle reste vraie m+1

$$D^1_{RD}(\mathfrak{D}^m_{RD}(F)) \setminus \mathfrak{D}^m_{RD}(F) = \emptyset$$

par propriété 1 on a

$$D^{m+1}_{RD}(F) \setminus \mathfrak{D}^m_{RD}(F) = \emptyset$$

or par hypothèse de récurrence on a

$$\mathfrak{D}^m_{RD}(F) = \bigcup_{i=o}^{n} D^i_{RD}(F) = \bigcup_{i=o}^{n} D^i_{RD}(F) = \mathfrak{D}^n_{RD}(F)$$

Dès lors,

$$\mathcal{D}^{m+1}_{RD}(F) \setminus \mathcal{D}^{n}_{RD}(F) = \emptyset$$

et donc $\mathcal{D}^{m+1}_{RD}(F) \subset \mathcal{D}^{n}_{RD}(F)$

2.2.1.6 Problème auxiliaire.

2.2.1.6.1 Spécification.

Etant donné que F est inclus dans FI et que FI contient tous les faits déductibles à un niveau à partir de $FI \setminus F$ et RD, calculer les nouveaux faits déductibles à un niveau à partir de FI et RD. Ce programme auxiliaire sera appelé PA(F)

Soient FI, RD globales

F un ensemble de faits tel que $F \subset FI$ et $D^1_{RD}(FI \setminus F) \subset FI$

PA(F) calcule $D^1_{RD}(FI) \setminus FI$.

2.2.1.6.2 Construction de PA(F).

Montrons, dans un premier temps, que

Si $D^1_{RD}(FI \setminus F) \subset FI$, $F \subset FI$

alors $D^1_{RD}(FI) \setminus FI = N \setminus FI$

où $N = \{ f \in \mathcal{F}$: Il existe une règle "$A \leftarrow B_1 \wedge \ldots \wedge B_k$" $\in RD$

une substitution θ

tel que - pour tout $1 \leqslant i \leqslant k$ $B_i\theta \in FI$.

- il existe $1 \leqslant j \leqslant k$ $B_j\theta \in F$
- $f = A\theta\zeta$

En effet,

$N \subset D^1_{RD}(FI)$ car $D^0_{RD}(FI) = FI$, $F \subset FI$ et par définition de N et de $D^1_{RD}(FI)$

Il suffit donc de montrer que

$$D^1_{RD}(FI) \setminus N \subset FI$$

$D^1_{RD}(FI) \setminus N = \{ f \in \mathcal{F}$: il existe une règle "$A \leftarrow B_1 \wedge \ldots \wedge B_k$" $\in RD$

une substitution θ

tel que - pour tout $1 \leqslant i \leqslant k$ $B_i\theta \in FI \setminus F$

- $f = A\theta\zeta$

$= D^1_{RD}(FI \setminus F) \subset FI$ par hypothèse et donc le théorème est démontré.

Cette propriété nous suggère un algorithme qui calculera cet ensemble N et lui retirera l'ensemble FI.

L'algorithme calculant N est le suivant

$N \leftarrow \emptyset$
<u>pour chaque</u> $f \in F$
  <u>pour chaque</u> "$A \leftarrow B_1 \wedge \ldots \wedge B_k$" $\in RD$
    <u>pour chaque</u> $1 \leqslant i \leqslant k$
      <u>Si</u> B_i et f s'unifient $(B_i\sigma = f)$
      <u>alors</u>
        <u>pour chaque</u> substitution φ tel que
        $\{B_1\sigma\varphi, \ldots, B_{i-1}\sigma\varphi, B_{i+1}, \ldots, B_k\sigma\varphi\} \subseteq FI$
        <u>faire</u> $N \leftarrow N \cup \{A\sigma\varphi\}$

Remarquons que dans cet algorithme, nous aurons toujours que $B_i\sigma = B_i\sigma\varphi = f$ car $B_i\sigma$ est totalement instantié (ne contient pas de variables) et que $\sigma\varphi$ est équivalente à la substitution θ de la définition. Cette substitution ne peut être directement obtenue en unifiant B_i et f. De plus, étant donné une substitution σ, il est possible de déduire plusieurs faits

<u>exemple</u> : $RD = \{ P(x,y) \leftarrow q(x) \wedge R(y) \}$
$FI = \{ Q(a), Q(b), R(c) \}$
$F = \{ R(c) \}$

l'unification de R(c) et R(y) donne $\sigma = \{(c/y)\}$ et nous pourrons trouver deux substitutions $\varphi_1 = \{(a/x)\}$
$\varphi_2 = \{(b/x)\}$
tel que $Q(x)\varphi_1$ et $Q(x)\varphi_2 \in FI$
Ces deux substitutions génèreront les deux faits

$P(x,y)\sigma\varphi_1 = P(a,c)$
$P(x,y)\sigma\varphi_2 = P(b,c)$

L'algorithme calculant $N \setminus FI$ est le suivant

- calcul de N
- $RES \leftarrow \emptyset$
- <u>Pour tout</u> $f \in N$
  Si $f \notin FI$ <u>alors</u>
  $RES \leftarrow RES \cup \{f\}$

et la fin de l'algorithme, RES contient $N \setminus FI$

Une autre alternative serait de réunir ces deux algorithmes en un seul qui testerait prématurément l'appartenance à FI du fait généré

Si B_i et f s'unifient ($B_i \sigma = f$)
alors dès que nous obtenons une substitution ϑ_1 telle que
- $(B_1 \sigma \vartheta_1, \ldots, B_k \sigma \vartheta_1) \subset FI$
- $A \sigma \vartheta_1$ est totalement instantié
  Si $A\sigma\vartheta_1 \in FI$ alors
  la recherche pour cette substitution ϑ_1 peut être arrêtée
  Sinon continuer comme dans l'algorithme vu précédemment.

Après l'exécution de cet algorithme, nous aurons directement l'ensemble recherché. En programmation logique, on sépare généralement les antécédants d'une clause en deux catégories : ceux qui contribuent à la génération de la solution et ceux quitestentla solution proposée. Cet algorithme sera d'autant plus efficace que le nombre de conditions est grand. Cependant, l'interaction avec la base de données se fera formule atomique par formule atomique contrairement à l'algorithme précédent et il sera nécessaire de tester constamment si $A\sigma\vartheta_1$ est totalement instantiée ce qui est une opération assez coûteuse.

2.2.1.7 Insert(f) : cas : $f \in FE$ et $f \in FI \setminus FE$.

2.2.1.7.1 Algorithme.

Si $f \notin FE$ et $f \in FI$ alors
$FE \leftarrow FE \cup \{f\}$

2.2.1.7.2 Justification.

Si $f \in FE$, le programme n'effectue aucune opération ce qui est équivalent à effectuer

BDDS ← BDDS ou encore
BDDS ← Insert(BDDS, f)

Si $f \in FI \setminus FE$, alors le programme effectue l'affectation

$\langle FE, RD, FI\rangle \leftarrow \langle FE \cup \{f\}, RD, FI\rangle$

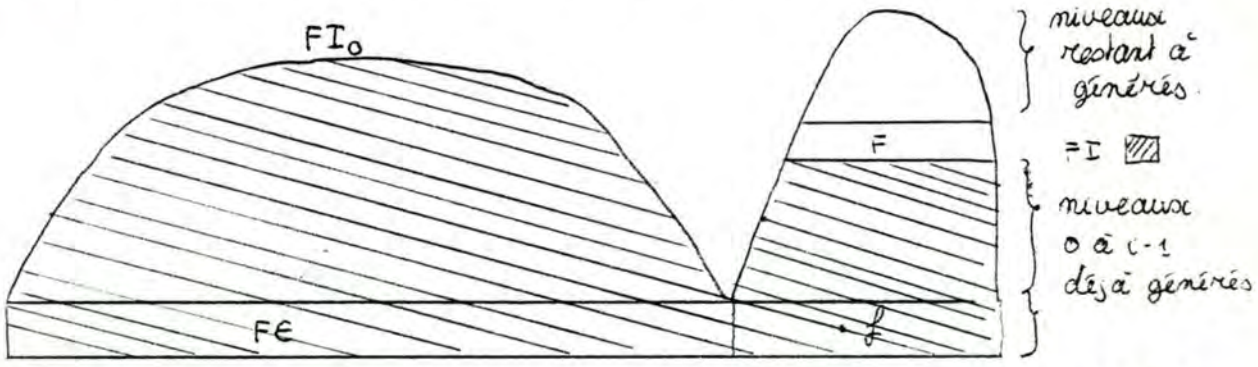
c.à.d.

BDS ← Insert(BDDS, f).

2.2.1.8 Insert(f) : cas : f ∉ FI.

2.2.1.8.1 Description de la situation initiale.

D'après les principes de la stratégie exposée précédemment, nous obtenons la situation générale suivante



- F contient les nouveaux faits directement déductibles à partir du FI initial et des nouveaux faits générés aux niveaux 0 jusque i.
- FI contient les faits implicites déductibles à partir de FE et RD ce qui constitue le FI initial noté FI_o ainsi que les nouveaux faits générés aux niveaux 0,...,i.

Plus précisément

$$FI = FI_o \cup D^o_{RD}(FI_o \cup \{f\}) \cup D^1_{RD}(FI_o \cup \{f\}) \quad \ldots \quad D^i_{RD}(FI_o \cup \{f\})$$

$$FI = FI_o \cup \mathcal{D}^i_{RD}(FI_o \cup \{f\}) = \mathcal{D}^i_{RD}(FI_o \cup \{f\}) \text{ si } i \geqslant o.$$

$$F = D^{i+1}_{RD}(FI_o \cup \{f\}) \setminus \mathcal{D}^i_{RD}(FI_o \cup \{f\})$$

par la propriété

$$F = D^1_{RD}(FI) \setminus FI.$$

Nous obtenons alors l'invariant

$$I = \exists\, i \geqslant o \quad FI = \mathcal{D}^i_{RD}(FI_o \cup \{f\})$$

$$F = D^1_{RD}(FI) \setminus FI.$$

2.2.1.8.2 Passage de la situation initiale à l'invariant.

Situation initiale :
- $FI = FI_o$
- $f \notin FI$
- FI_o sont les faits implicites associés à FE et RD (car BDDS est une base de données déductive saturée, ce qui nous assure que $D^1_{RD}(FI_o) \subset FI_o$).

Si nous effectuons les opérations

$F \Leftarrow \{f\}$

$FI \leftarrow FI \cup \{f\}$

nous aurons que $F \subset FI$ et $D^1_{RD}(FI \setminus F) = D^1_{RD}(FI_o) \subset FI$,

car $FI_o \subset FI$

$D^1_{RD}(FI_o) \subset FI_o$

Nous pourrons faire

$F \leftarrow PA(F)$

Pour obtenir, d'après la spécification de PA(F)

$F = D^1_{RD}(FI) \setminus FI$

$FI = FI_o \cup \{f\} = \mathcal{D}^o_{RD}(FI_o \cup \{f\})$.

L'invariant est donc vérifié pour I=o.

2.2.1.8.3 Test de fin de boucle.

Selon la stratégie utilisée, tous les nouveaux faits seront générés lorsque F sera vide. En effet,

Si il existe $i \geq o$ $FI = \mathcal{D}^i_{RD}(FI_o \cup \{f\})$

$F = D^1_{RD}(FI) \setminus FI = \emptyset$

alors soit k la valeur de i vérifiant I.

D'après la propriété 3, nous aurons que

$\forall m > k \quad D^m_{RD}(FI_o \cup \{f\}) \subset \mathcal{D}^k_{RD}(FI_o \quad f) = FI$[5]

c.à.d. que $FI = \mathcal{D}^\infty_{RD}(FI_o \cup \{f\})$.

FI est donc l'ensemble des faits implicites associés à $FI_o \cup \{f\}$ et RD. Comme FI_o est l'ensemble des faits implicites associés à FE et RD, FI est l'ensemble des faits implicites associés à $FE \cup \{f\}$ et RD.

Il ne restera alors qu'a effectuer l'affectation

$FE \leftarrow FE \cup \{f\}$

et le programme aura effectué l'affectation

$(FE, RD, FI) \leftarrow (FE \cup \{f\}, RD, FI')$

ou FI' est l'ensemble des faits implicites associé à $FE \cup \{f\}$ et RD

c.à.d. l'affectation

BDDS ← insert(BDDS, f).

2.2.1.8.4 Passage de la situation générale à la situation générale

D'après la stratégie utilisée, lorsque F est non vide, il suffira de l'inclure à FI et de calculer les nouveaux faits directement déductible de FI et RD. Si nous avons

$$I = \exists i \geq o \; FI = \mathcal{D}^{i}_{RD}(FI_o \cup \{f\})$$

$$F = D^{1}_{RD}(FI) \setminus FI$$

et $F \neq \emptyset$

posons FI_1 la valeur de FI avant l'exécution de la boucle

F_1 la valeur de F avant l'exécution de la boucle

i_1 la valeur de i vérifiant I

Si nous effectuons l'opération

$FI \leftarrow FI \cup F$

Nous aurons

$$FI = FI_1 \cup F_1 = FI_1 \cup (D^{1}_{RD}(FI_1) \setminus FI_1)$$

$$= FI_1 \cup D^{1}_{RD}(FI_1)$$

et donc $- D^{1}_{RD}(FI)F) = D^{1}_{RD}(FI_1) \subset FI$

$- F \subset FI$

Nous pourrons alors appliquer PA(F)

$F \leftarrow PA(F)$

pour obtenir

$$F = D^{1}_{RD}(FI) \setminus FI$$

$$FI = FI_1 \cup D^{1}_{RD}(FI_1)$$

$$= \mathcal{D}^{il}_{RD}(FI_o \cup \{f\}) \cup D^{1}_{RD}(\mathcal{D}^{il}_{RD}(FI_o \setminus \{f\}))$$

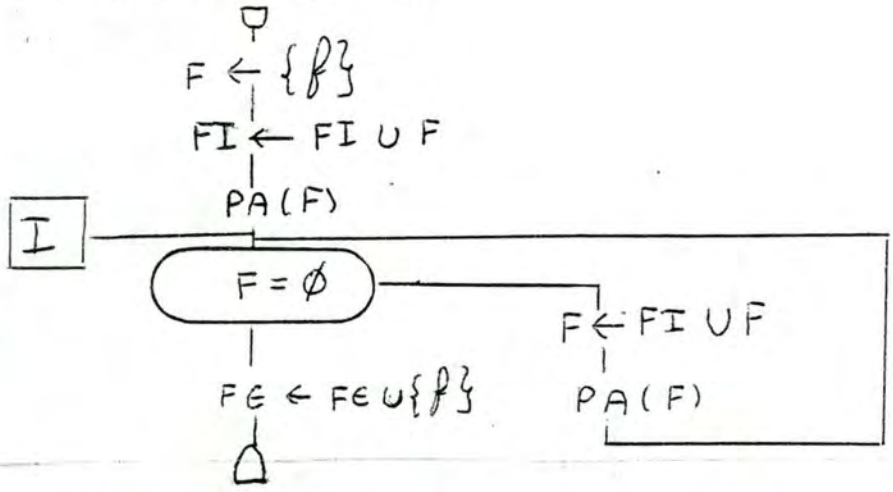
et par la propriété 2

$$= \mathcal{D}^{il+1}_{RD}(FI_o \cup \{f\})$$

Dès lors, si l'invariant est vrai pour i_1 il est vrai pour $i = i_1+1$.

2.2.1.8.5 Algorithme.

L'algorithme sera donc



2.2.1.8.6 Terminaison.

Soit une fonction $\alpha(FI) = n - \#FI$

où $n = \#\mathcal{F}$ où $\mathcal{F}$ est l'ensemble fini de faits

où $\#A$ est le nombre d'éléments de l'ensemble A.

Nous aurons toujours

$FI \subset \mathcal{F}$ et donc $n - \#FI \geq 0$.

c.à.d. pour toute valeur de FI, $\alpha(PI) \geq 0$.

Si FI_1, F_1 sont les valeurs de FI et de F avant l'exécution du corps de la boucle.

FI_2, F_2 sont les valeurs de FI et F après exécution du corps de la boucle

nous aurons que $\alpha(FI_2) < \alpha(FI_1)$

En effet, nous aurons que $F_1 \neq \emptyset$

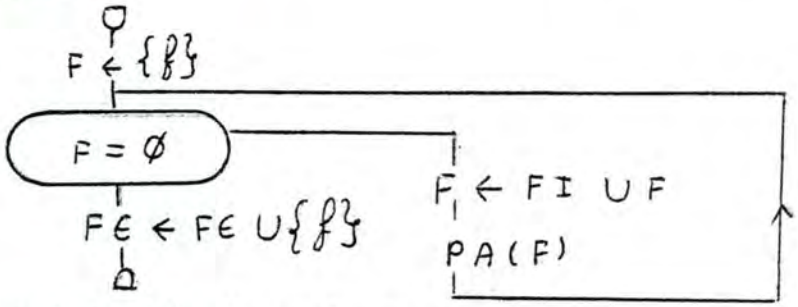
$$F_1 = D^1_{RD}(FI_1) \setminus (FI_1)$$
$$= F_1 \cap FI_1 \neq \emptyset$$

Dès lors, nous aurons $FI_2 = FI_1 \cup F_1$

$$\alpha(FI_2) = \alpha(FI_1 \cup F_1)$$
$$= n - \#(FI_1 \cup F_1)$$
$$= n - \#FI_1 - \#F_1 + \#(FI_1 \cap F_1)$$
$$= \alpha(FI_1) - \#F_1 + 0$$
$$< \alpha(FI_1) \quad \text{car } \#F_1 > 0 \text{ car } F_1 \neq \emptyset$$
$$\text{et } (FI_1 \cap F_1) = 0 \text{ car } FI_1 \cap F_1 = \emptyset$$

2.2.1.8.7 Algorithme final.

Une simple transformation de programme permet d'obtenir le programme suivant



2.2.1.9 Complexité théorique de Insert.

La complexité théorique de cet algorithme peut difficilement être évaluée. En effet, le nombre de nouveaux faits implicites ne dépend pas du nombre de faits implicites initial ni du nombre de règles de déduction mais bien de la nature de ces règles et des faits implicites.

exemple : RD = $Q(x,y) \leftarrow P(x,y) \wedge R(x)$

FI = $\{ P(1,1), P(1,2), P(1,3), \ldots, P(1,10\ 000) \}$

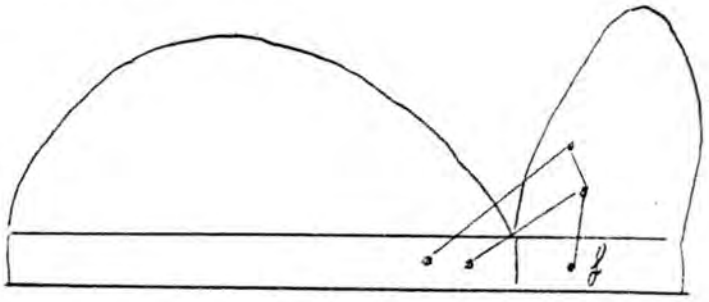
Si f = R(1) alors le nombre de nouveaux faits implicites sera 10 000 alors que si f = R(2) il n'y aura pas de nouveaux faits implicites. Une mesure intéressante de la complexité de cet algorithme serait le nombre d'accès à la base de données effectués. Mais, encore une fois, ce nombre ne dépend ni du nombre de faits implicites initial ni du nombre de règle de déduction. Un accès BD correspond dans notre algorithme à tester si $B_i\theta \in$ FI ou à effectuer FI = FI $\cup \{ f_i \}$
Le premier type d'accès correspond à une question alors que le second type correspond à l'ajout d'un élément à la BD.
Nous remarquerons cependant que notre algorithme n'effectue pas une recherche "aveugle" de nouveaux faits implicites mais la recherche proprement dite (algorithme PA) est guidée par le choix des règles contenant les nouveaux faits.
Une approche pour étudier la complexité théorique consisterait à calculer celle-ci à partir d'une BDDS déterminée ou du moins à partir d'informations statistiques au sujet de celle-ci. Une étude intéressante serait de spécifier ces informations statistiques ainsi que des règles de calcul de la complexité par rapport à ces informations statistiques.

2.2.1.10 Suppress(f)

Dans les paragraphes qui précèdent nous avons présenté de manière détaillée la construction de l'algorithme d'insertion. Dans cette partie, pour des raisons de temps et de place, nous n'avons pas jugé opportun de présenter, de manière approfondie, l'algorithme de suppression. Nous nous limiterons à une présentation intuitive de la stratégie appliquée tout en ayant conscience que les informations données sont incomplètes et devraient être détaillées pour obtenir une justification satisfaisante.

Nous étudierons le cas où $f \in FI$.

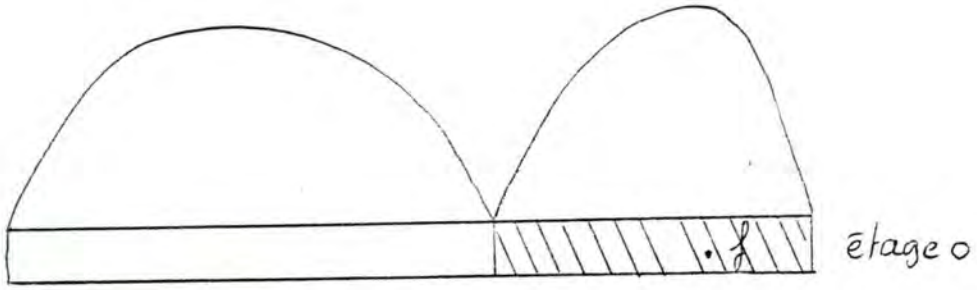
2.2.1.10.1 Stratégie.



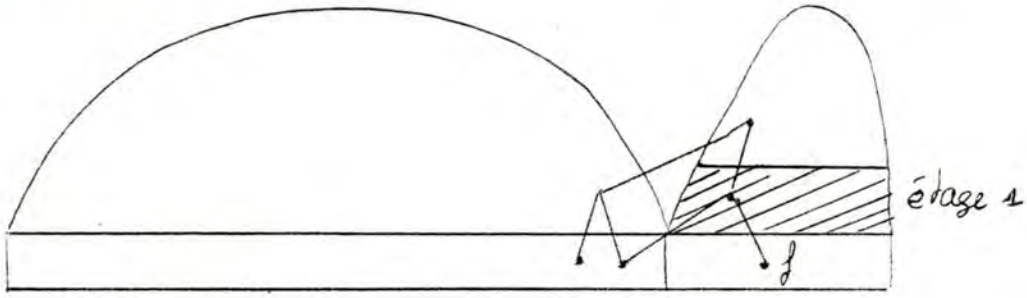
La stratégie consiste à parcourir l'ensemble des conséquences logiques selon une succession d'étages. Chaque étage correspond aux faits implicites directement déductibles à partir de (FI moins les étages inférieurs) auxquels on a retiré les conséquences logiques de $FE \setminus \{f\} \cup RD$. Cette stratégie est inapplicable comme telle puisqu'elle suppose la connaissance des conséquences logiques de $FE \setminus \{f\} \cup RD$ qui constitue le résultat cherché.

En fait, un résultat équivalent peut être obtenu en déterminant si un fait implicite susceptible d'être supprimé peut être déduit d'une manière différente auquel cas il est alors une conséquence logique de $FE \setminus \{f\} \cup RD$.

Une solution à ce problème [NICO 83] consiste à associer à chaque fait de la BDDS le nombre de manières différentes dont il peut être déduit. Nous appellerons ce nombre, compteur implicite du fait. Dans ce qui suit, nous ferons l'hypothèse que les règles ne sont pas récursives ou mutuellement récursives et nous prendrons comme convention que, pour un fait explicite, le fait qu'il soit automatiquement implicite sans applications des règles de déduction ne soit pas comptabilisé dans le compteur implicite de ce fait.



L'étage O est constitué de f si ce fait ne peut être déduit de FE\{f} c.à d. si son compteur implicite est nul. Sinon, l'étage O est l'ensemble vide



L'étage 1 se compose des faits implicites directement déductibles de FI nécessitant la présence de l'étage O si ceux-ci ont un compteur implicite égal à 1. Dans ce cas, ceux-ci doivent être supprimés car ils ne sont plus conséquences de $(FE \setminus f) \cup RD$.
L'étage i se compose des faits directement déductibles à partir de (FI\(étage O ∪ étage 1 ∪ ... ∪ étage i-2) nécessitant la présence de l'étage i-1 et qui ont un compteur implicite égal à 1.
Les étages seront ainsi supprimés successivement jusqu'à ce que l'on obtienne un étage vide puisque, dans ce cas, les étages supérieurs seront également vides.
L'invariant sera donc

$FI = FI_o \setminus (\text{étage O} \cup \text{étage 1} \cup \ldots \cup \text{étagei})$
F = étage i

NB. Dans ce qui suit, nous supposerons que lorsqu'un fait peut être n ($n \geq 1$) manières différentes, il existe n faits implicites distincts représentant le même fait pour ce qui est de l'appartenance à l'ensemble.

L'initialisation sera obtenue de la manière suivante
Si le compteur implicite de f = O alors $F \leftarrow \{f\}$; $FI \leftarrow FI \setminus \{f\}$
Sinon $F = \emptyset$

Le test de fin est

$F = \emptyset$

La boucle est composée des instructions suivantes

(1) $F \leftarrow$ ensemble des faits directement déductibles à partir de $FI \cup F$ nécessitant la présence de F

(2) $\forall f \in F$

diminuer de 1 son compteur implicite

(3) $F \leftarrow \{f \mid f \in F$ et compteur implicite de $F = 0\}$

Après ces opérations, F contient les faits implicites directement déductibles de ($FI_o \setminus$ étage $0 \cup$ étage $1 \cup \ldots \cup$ étage i-1) nécessitant la présence de l'étage i, ce qui entraine que F = étage i+1. De plus, FI = $FI_o \setminus$ (étage $1 \cup$ étage $2 \cup \ldots \cup$ étage i-1) en vertu de la seconde assignation et l'invariant est donc vérifié.

Il sera cependant nécessaire de modifier l'algorithme d'insertion pour tenir compte de ces compteurs implicites. C'est ainsi que lors du calcul de PA(F), nous trouverons trois étapes

1) Calcul de N : fait directement déductible de FI F
2) Calcul de $N \setminus FI$ donnant F
3) Pour tout $f \in N \cap FI$
   augmenter de 1 le compteur implicite de f.

2.2.1.10.2 Limitations.

Il est possible de lever les hypothèses concernant la récursivité des règles. Cependant, dans ce cas, le processus ne répondra plus à la spécification. Considérons par exemple, les règles

$R(X) \leftarrow T(X)$
$S(X) \leftarrow R(X)$
$T(X) \leftarrow S(X)$

Après l'insertion de T(a) la BDDS contient T(a), S(a) et R(a) et leurs compteurs implicites respectifs sont 2,1,1. La suppression de T(a) laisse la base de données dans l'état

T(a), S(a) et R(a)

ce qui n'est pas un ensemble de faits déductibles de $\emptyset$ et RD.

Le processus fonctionnera uniquement si nous assurons qu'aucun fait n'est déductible de lui-même. Il est également possible de construire un algorithme qui prenne ce cas en considération mais il entraine une dégradation notable des performances.

2.2.1.10.3 Autres alternatives.

Vu les limitations énoncées, on peut se demander s'il n'existe pas d'autres approches possibles. Nous en avons relevé trois.
La première consiste à resaturer FE \{f}.
La seconde consiste, en présence d'un fait f_1 susceptible d'être supprimé, de tester s'il peut être déduit d'une manière différente. Cela revient à trouver une règle $A \leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_n$

telle que
- $f_1 = A\theta$
- $B_i\theta \in$ FI
- chaque fait B_i est explicite ou peut être déduit des faits explicites et des règles déduction.

La troisième approche consiste à garder toutes les traces de déduction pour chaque fait implicite.

2.2.1.10.4 Conclusion.

L'approche avec compteur implicite est de loin la plus efficace mais ne peut répondre à la spécification dans certains cas de règles. Dès lors, soit on restreint la classe des règles, soit on admet dans certains cas que la base de données ne soit pas dans un état cohérent. En ce qui concerne les alternatives proposées, la première est manifestement inefficace au point de vue temps d'exécution. La seconde parait plus attrayante mais elle demande de réaliser deux processus de déduction, ce qui sera également coûteux en temps. Enfin, la dernière est inacceptable pour ce qui est de la place mémoire. Nous conseillons par conséquent, d'adopter la première solution en restreignant la classe des règles considérée.

2.2.2 L'approche en dérivation.

2.2.2.0 Introduction.

Contrairement à l'approche en génération qui utilisait les règles de déduction pour saturer la base de données (c.à.d. pour générer tous les faits qui étaient des conséquences logiques des règles de déduction et des faits élémentaires), l'approche en dérivation ne mémorise que les faits explicites et utilise les règles de déduction pour dériver les conséquences logiques des axiomes et des faits élémentaires lorsque cela s'avère nécessaire.
Les règles de déduction sont utilisées dans cette approche comme les théorèmes "conséquents" de Planner [HEWI 69].

Il est traditionnel, dans cette approche, de séparer la base de données déductive en deux bases de données : la base de données extensionnelle qui contient tous les faits élémentaires et qui peut être, par exemple, une base de données relationnelle et une base de données intentionnelle qui contient l'ensemble des règles de déduction.
Dans une base de données déductive, une relation peut être définie soit uniquement dans la base de données extensionnelle soit dans les deux bases de données. Cette dernière possibilité n'apporte aucune puissance d'expression supplémentaire. Toute relation définie extensionnellement et intentionnellement peut être redéfinie par un ensemble de relations qui sont définies dans une et une seule des deux bases de données.

L'approche en dérivation a donné lieu à deux méthodes, la méthode compilée et la méthode interprétée selon que l'on sépare ou non les accès à chacune des deux bases de données. Dans l'approche interprétée, on interrogera simultanément avec les deux bases de données. Cette approche a été étudiée, par exemple, dans [MINK 78] [WARR 81].
La méthode compilée sépare, quant à elle, l'accès à la base de données intentionnelle de l'accès à la base de données extensionnelle. Elle est étudiée, par exemple, dans [CHAN 78] [REIT 78].

Cependant, quelle que soit la méthode choisie, les règles de déduction récursives posent des problèmes. De nombreux auteurs s'y sont intéressés. [HENS 84], [CHAN 81] , [LOZI 84] , [VIEI 85], [MINK 82a] . Bien que nous nous placions plutôt dans l'optique "bases de données déductives" les raisonnements développés ci-dessous restent valables pour les bases de données logiques. Dans ce cas une partie du système sera dédicacée au traitement des règles de déduction et à l'interaction avec la base de données relationnelle.

2.2.2.1 La méthode interprétée.

La méthode interprétée ne fait aucune distinction entre l'accès à la base de données extensionnelle et la base de données intentionnelle. L'objectif de cette approche était de pouvoir utiliser les résultats en démonstration automatique ; les démonstrateurs auraient alors été adaptés à la manipulation d'ensembles ce qui peut se faire de deux façons distinctes : soit en manipulant des ensembles de réponses en chaque noeud de l'arbre de résolution soit en utilisant des piles pour mémoriser l'ensemble des réponses qui seront alors obtenues par backtracking [CHAK 79]. Cette dernière solution semble plus adéquate pour les bases de données logiques. Dans la méthode interprétée, dès que l'on se trouve en présence d'un littéral qui fait référence à la base de données extensionnelle, on interagit avec la base de données relationnelle pour obtenir un ensemble de solutions. Ces solutions seront, en principe, utilisées pour résoudre les autres littéraux. Cependant, contrairement à la programmation en logique, aucune information de contrôle n'est fournie dans une base de données déductive. Dès lors, tout le contrôle doit être assuré par le système de gestion de la base de données et cela pose essentiellement deux problèmes.

Le problème d'optimisation.

L'unité d'accès à la base de données relationnelle étant la relation, il n'est pas possible d'exploiter les mécanismes d'optimisation de questions mis en oeuvre dans le cadre de ces bases de données. En conséquence, la méthode interprétée devra assurer, elle-même, cette optimisation. Ces optimisations seront prises en compte par l'intermédiaire de la fonction de sélection qui déterminera quel est le littéral le plus intéressant à résoudre compte tenu, d'une part, de l'instantiation des variables et, d'autre part, de la structure physique de la base de données (eg : index, taille des relations etc.). Des exemples de telles fonctions peuvent être trouvées dans [WARR 81], [MINK 78] .

Le problème des règles récursives.

En programmation logique, la terminaison du programme est sous la responsabilité du programmeur. Dans le cas des bases de données déductives, le problème de terminaison sera reporté sur le SGBD.

exemple : Considérons les règles de déduction

$A(x,y) \leftarrow F(x,y)$

$A(x,y) \leftarrow P(x,z) \wedge A(z,y)$

En supposant que l'extension des relations F et P soient données par

F		
	a	g
	c	d

P		
	a	b
	b	c
	c	a

et que la question soit "A(a,y)"

l'arbre de déduction (figure 1) contiendra une branche infinie. Comme il est nécessaire de ramener toutes les solutions, le démonstrateur ne se terminera jamais.

Remarquons, cependant, que dans une base de données déductive, d'une part, le nombre de constantes est fini et, d'autre part, nous avons exclu les symboles de fonction des règles de déduction. Par conséquent, il devrait exister un algorithme qui permette de déterminer toutes les réponses à une question queiconque. L'approche en génération indique d'ailleurs qu'un tel algorithme existe.

On peut alors envisager d'implémenter des détecteurs de boucles dans les démonstrateurs de théorèmes. C'est ainsi que la branche infinie de la figure 1 pourra être coupée en considérant que la question A(a,RES) a déjà été posée. Diverses techniques ont été envisagées dans [WALK 81] mais aucune n'est à la fois complète et cohérente. Il semble, par conséquent, que la solution repose sur des algorithmes spécialement conçus pour ce cadre plutôt que sur l'adaptation des résultats obtenus en démonstrations de théorèmes.
Au moment où ce mémoire était en cours d'impression, nous avons pris connaissance d'une méthode extrêmement intéressante.
Cette méthode [LOZI 84] utilise, pour résoudre le problème des règles récursives, une résolution ascendante et descendante simultanément. Les règles de déduction sont utilisées, à la fois, en génération pour produire des faits qui sont pertinents à la question et en dérivation pour faire apparaître des sous-questions à partir de la question initiale, sous-questions qui sont résolues récursivement. Cette approche est aussi appelée "Query Subquery". Cette méthode présente cependant quelques problèmes d'optimisation et différentes solutions sont pro-

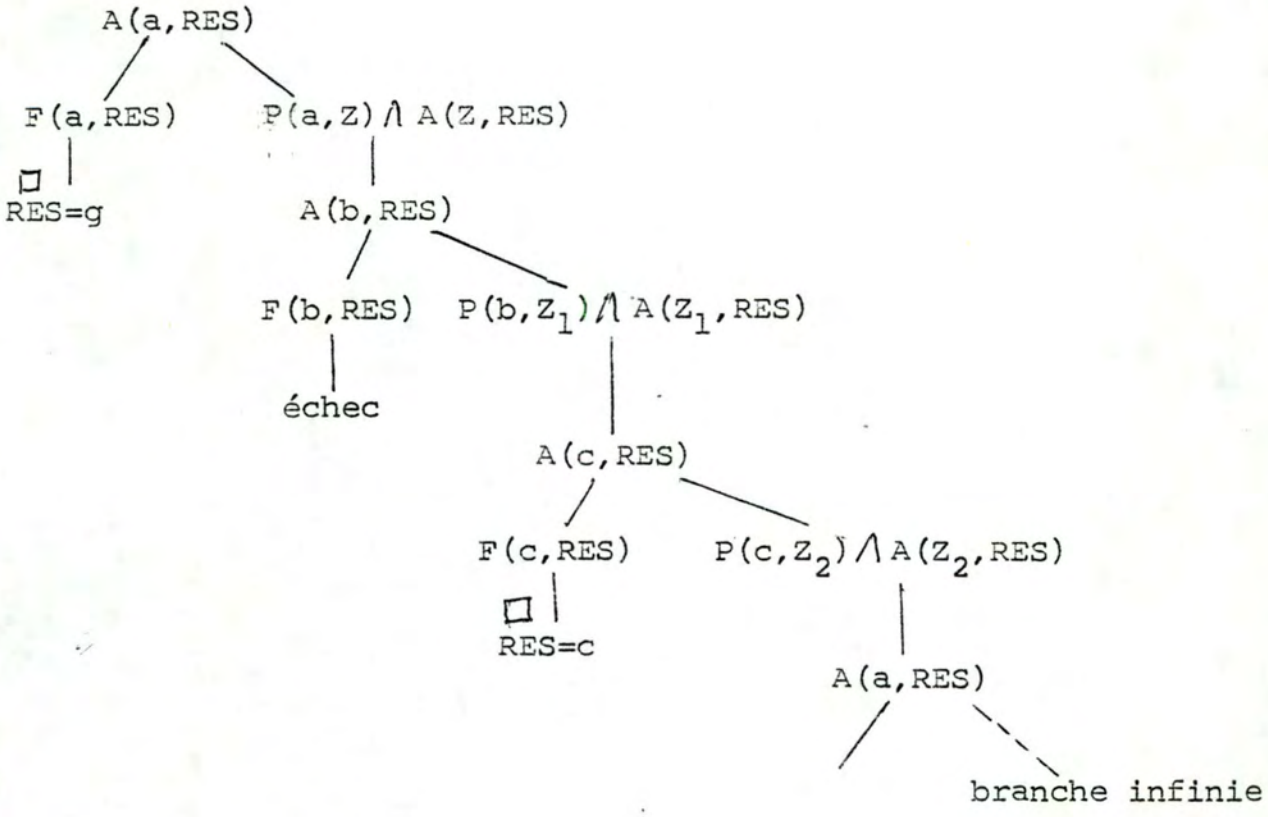

A(a,RES)
F(a,RES)
P(a,Z) ∧ A(Z,RES)
RES=g
A(b,RES)
F(b,RES)
P(b,Z1) ∧ A(Z1,RES)
échec
A(c,RES)
F(c,RES)
P(c,Z2) ∧ A(Z2,RES)
RES=c
A(a,RES)
branche infinie

Figure 1

posées dans [VIEI 85].
Ces méthodes semblent relativement satisfaisantes mais de nombreuses optimations restent possibles.

2.2.2.2 La méthode compilée.

La méthode compilée, quant à elle, sépare l'interaction avec la base de données extensionnelle de l'interaction avec la base de données intentionnelle.
L'objectif poursuivi, par cette méthode, était l'utilisation des optimisations de questions développées dans le cadre des bases de données relationnelles.
Pour ce faire, cette méthode compilait un ensemble initial de règles de déduction en un autre ensemble où les antécédents des clauses étaient uniquement des relations définies dans la base de données extensionnelle. Cette compilation peut n'être réalisée qu'une fois, par exemple lors de la construction du schéma de la base de données.

exemple : Les règles de déduction (où R,V,W sont des relations de la base de données extensionnelle)

$P(x,y) \leftarrow Q(x,z) \wedge R(z,y)$
$Q(x,y) \leftarrow T(x,z) \wedge V(z,y)$
$Q(x,y) \leftarrow W(x,y)$
étaient compilées en
$P(x,y) \leftarrow T(x,z_1) \wedge V(z_1,z) \wedge R(z,y)$
$P(x,y) \leftarrow W(x,z) \wedge R(z,y)$
$Q(x,y) \leftarrow T(x,z) \wedge V(z,y)$
$Q(x,y) \leftarrow W(x,y)$

La question P(x,a) donne lieu aux questions suivantes soumises au SGBD
(1) $T(x,z_1) \wedge V(z_1,z) \wedge R(z,a)$
(2) $W(x,z) \wedge R(z,a)$

Les questions ainsi déterminées pourront bénéficier des optimisations réalisées dans le cadre des bases de données relationnelles. Cependant tout comme dans la méthode interprétée, nous mentionnerons deux problèmes.

Le problème des règles récursives.

Les règles récursives auxquelles on appliquera la compilation défi-

nie ci-dessus conduiront à une séquence infinie de clauses. En effet, considérons les règles de déduction

$A(x,y) \leftarrow ami(x,y)$

$A(x,y) \leftarrow père(x,z) \wedge A(z,y)$

où

ami(x,y) est vrai si x est un ami de y

père(x,y) est vrai si y est le père de x

où ami et père sont des noms de relations définis dans la BD extensionnelle.

La compilation de ces clauses donne lieu à la génération de la séquence infinie :

(1) $A(x,y) \leftarrow ami(x,y)$

(2) $A(x,y) \leftarrow père(x,z) \wedge ami(z,y)$

(3) $A(x,y) \leftarrow père(x,z_1) \wedge père(z_1,z) \wedge ami(z,y)$

(4) $A(x,y) \leftarrow père(x,z_2) \wedge père(z_2,z_1) \wedge père(z_1,z) \wedge ami(z,y)$

⋮

Par conséquent, la compilation de ces clauses donnera lieu à la génération d'un programme itératif dans le cadre des bases de données déductives. A chaque passage dans la boucle, on évaluera une clause de la séquence. Dans le cadre d'une base de données logique, un algorithme générera chacune des clauses et les transmettra au SGBD pour être évaluée. Dans les deux approches, le problème est de déterminer la condition d'arrêt soit du programme généré (bases de données déductives) soit de l'algorithme de génération et évaluation (base de données logique). La simple condition d'arrêt "la clause i ne produit plus de réponse" ne peut être considérée comme une condition adéquate car un individu peut avoir un grand-père qui n'a pas d'amis et un arrière grand-père qui en a. Le lecteur intéressé par la condition d'arrêt tout à fait générale peut se reporter à [VIEI 85]. Cependant, dans de nombreux cas, il est possible de tenir compte des propriétés sémantiques de la relation pour améliorer cette condition d'arrêt. Il faudrait, dès lors, fournir un peu plus d'informations que les clauses.

Le problème d'optimisation.

La séquence infinie de clauses fait apparaître de nombreuses redondances et il est naturel de penser, lors de l'évaluation d'une clause i, à utiliser les résultats obtenus lors de l'évaluation de la clause i-1. Ainsi par exemple, la clause (4) permet d'utiliser l'évaluation de "$P(x,z_1)$ $P(z_1,y)$" obtenue lors de l'étape précédente. Ce problème

d'optimisation semble être particulièrement difficile à traiter dans cette méthode de sorte que la méthode interprétée parait plus adéquate, du moins pour les bases de données logiques.
En ce qui concerne les bases de données déductives, différents travaux ont été réalisés. Chang [CHAN 79] présente une solution dans le cas où les clauses sont régulières (un seul littéral défini dans la base de données extensionnelle). Ces travaux sont d'ailleurs étendus dans [MINK 82] mais la classe de clauses considérée reste limitée.
Les travaux les plus avancés sont, à notre connaissance, ceux de Henshen et Naqui [HENS 84]. Il utilise, en effet, une technique pour éviter les évaluations redondantes. Cependant, leur approche présente quelques problèmes. Les optimisations et la condition d'arrêt ne sont pas valables pour tout type de clauses (une telle condition d'arrêt est définie dans [VIEI 85]). De plus, le programme itératif est généré non seulement à partir des règles de déduction mais aussi à partir de la forme de la question. Si l'on veut permettre de poser n'importe quelle question, il sera nécessaire de mémoriser une grande quantité de programmes.
Les résultats obtenus dans ce cadre sont par conséquent loin d'être satisfaisants.
Il faudrait s'efforcer de lever la contrainte qui consiste à partir de la question et de généraliser les techniques d'optimisation.

2.3 Conclusion.

Nous avons donc défini les bases de données déductives et montré comment elles pouvaient être mises en oeuvre. En guise de conclusion, nous voudrions aborder un sujet qui n'est presque jamais mentionné dans la littérature : les motivations qui ont conduit au développement des bases de données déductives. Les bases de données déductives possèdent une théorie bien définie mais on peut se demander quels sont les objectifs pratiques poursuivis dans ces recherches. Il est clair que l'interconnection d'un langage de programmation avec une base de données peut permettre au premier cité de développer des applications de plus grande envergure. Mais quel est l'apport pratique pour le champ "base de données"? Nous avons relevé trois motivations qui peuvent être à la base de ces recherches.

1° Les bases de données déductives vont permettre d'économiser l'espace mémoire.

Il convient immédiatement de rectifier cet objectif. En effet, selon que l'on se place dans l'approche en dérivation ou dans l'approche en génération on économisera ou on n'économisera pas de la place mémoire. Ensuite, même si l'on se place dans l'approche en dérivation, l'espace mémoire économisé doit être contrebalancé avec le temps d'exécution supplémentaire qui pourrait augmenter. Cette motivation nous semble peu convaincante.

2° Les bases de données déductives offrent un nouveau moyen d'expression.

Remarquons immédiatement que le seul avantage pratique que pourraient apporter les bases de données déductives est la prise en compte de lois générales récursives. Sans cela, les lois générales peuvent être ramenées au concept de vue dans les bases de données relationnelles. Par conséquent, il est possible de poser deux questions

1° Les règles de déduction peuvent-elles être considérées comme des données ?

2° Existe-t-il suffisamment d'applications où ces lois s'avèreraient réellement utiles?

Nous relierons à la première question l'affirmation commune selon laquelle "la programmation logique abolirait la distinction entre les programmes et les données". En effet, les clauses de Horn peuvent

exprimer à la fois des données et des programmes. Cependant, nous avons mentionné dans la partie préliminaire, que le composant logique était un programme. Dès que l'on définit des règles de déduction récursives, et cela même si elles sont sans symbole de fonction, on peut considérer que l'on écrit un programme logique. La distinction entre programmes et données existerait donc toujours même si elles sont exprimées dans un même formalisme. Cependant, il est possible de considérer que dans un certain nombre de cas, les règles de déduction constituent une représentation commode pour une connaissance. Le problème est alors de savoir s'il existe suffisamment de situations pratiques où les règles de déduction s'avèrent utiles.

3° Les bases de données déductives permettent d'augmenter les puissances des langages d'interrogation.

Nous avons mentionné qu'un langage d'interrogation ne devait pas utiliser l'itération et la récursivité. Disposant d'une base de données qui contient la relation "Père(Parent,enfant)", il ne sera donc pas possible de calculer la relation ancêtre (paternel). Dans une base de données déductive cela sera possible en ajoutant deux règles de déduction et on peut par conséquent considérer que l'on a augmenté la puissance du langage d'interrogation lui permettant de calculer des fermetures transitives de relations tout en n'utilisant pas la récursivité et l'itération. Le problème sera donc de trouver quelles sont les questions fréquentes et intéressantes pour un ou plusieurs utilisateurs et d'exprimer les règles qui permettront d'y répondre.

CHAPITRE 3. LANGAGES D'INTERROGATION.

3.0 Introduction.

Après avoir étudié une contribution de la logique des prédicats du premier ordre au composant structuration, nous nous tournons vers le second composant, le composant manipulation. Ce chapitre étudiera, en effet, une contribution de la logique du premier ordre, par l'intermédiaire de la formalisation selon la vue "théorie du modèle", aux langages d'interrogation. Nous étudierons, dans un premier temps, la représentation des questions pour ensuite passer à la mise en oeuvre de ces questions. La représentation des questions concernera essentiellement le calcul relationnel à variable-tuple et à variable-domaine tandis que la mise en oeuvre présentera brièvement les optimisations syntaxiques et sémantiques basées sur la logique.

3.1 Représentation des questions.

3.1.1 Introduction.

Les langages d'interrogation de bases de données relationnelles peuvent être divisées en deux catégories

- les langages basés sur l'algèbre relationnelle
- les langages basés sur la logique des prédicats.

Dans ce chapitre, nous nous concentrerons sur la deuxième catégorie. Les langages d'interrogation constituent un des premiers impacts de la logique sur le champ "bases de données". On attribue généralement à [KUHN 67] l'idée selon laquelle la logique peut être utilisée comme langage d'interrogation. La raison en est qu'une base de données relationnelle peut être vue comme une interprétation d'un langage de premier ordre. C'est donc une contribution de la formalisation selon la vue "théorie du modèle" que nous étudierons ici.

Nous commencerons par étudier comment la logique des prédicats du premier ordre peut être modifiée pour définir un langage d'interrogation. Nous serons amenés à présenter deux langages

- le calcul relationnel à variable-tuple.
- le calcul relationnel à variable-domaine.

Comme l'algèbre relationnelle, ces deux langages constituent des langages d'interrogation abstraits en ce sens qu'ils ne seront jamais mis en oeuvre en tant que tels dans un système de gestion de base de

données. C'est pourquoi nous présenterons brièvement deux langages existants qui sont respectivement des mises en oeuvre du calcul relationnel à variable-tuple et du calcul relationnel à variable-domaine. Nous terminerons par une réflexion sur les types dans les langages d'interrogation.

3.1.2 La logique des prédicats comme langage d'interrogation.

Nous savons qu'une base de données peut être formalisée comme une interprétation d'un langage de premier ordre. De plus, dans ce cadre, le domaine d'interprétation sera fini puisqu'il n'existe qu'un ensemble fini de tuples dans les relations de la base de données et que, comme nous le verrons, ce seront ces tuples ou les composants de ces tuples qui seront les éléments du domaine d'interprétation. Dès lors, la valeur de vérité d'une formule peut être définie de manière plus simple. En effet, considérons une formule bien formée F. Sa valeur de vérité peut être définie de la manière suivante [Piro 78].

(1) remplacer chaque sous-formule de F de la forme
$\forall x\, A(x)$ par $A(a_1) \wedge \ldots \wedge A(a_m)$
où - A(x) indique que x est une variable libre dans A
- $a_1, \ldots, a_m$ sont tous les éléments de D
- A(x) représente la substitution de " a" pour toute occurence libre de x dans A

(2) remplacer chaque sous-formule de la forme
$\exists x\, A(x)$ par $A(a_1) \vee \ldots \vee A(a_m)$

(3) remplacer chaque variable restante x_j, qui est donc une variable libre, par une constante a_j.

(4) appliquer les fonctions à leurs arguments selon l'assignation effectuée pour les symboles de fonction

(5) appliquer les prédicats à leurs arguments selon l'assignation effectuée pour ces prédicats

(6) évaluer la proposition qui en résulte selon les règles habituelles de la logique des propositions. La valeur de la formule F est la valeur de vérité de cette proposition.

Cependant, avant que le langage de la logique des prédicats du premier ordre puisse être utilisée comme langage d'interrogation, il conviendra de préciser quel est le rôle des symboles de fonction et des variables libres et de définir quels sont les éléments du domaine d'interprétation. C'est ce que nous nous proposons de faire dans ce qui suit.

Les fonctions.-

En logique des prédicats du premier ordre, les fonctions expriment une dépendance entre un élément du domaine et un ou plusieurs autres éléments du domaine. Dans le contexte des bases de données relationnelles, ces dépendances sont exprimées par des dépendances fonctionnelles. Dès lors, il n'est pas nécessaire d'inclure dans le langage d'interrogation les symboles de fonction .

Les variables libres.

Selon la définition de la valeur de vérité d'une formule F, il est clair que F ne pourra prendre que deux valeurs "vrai" ou "faux". De plus, la valeur de vérité d'une formule F, qui contient une ou plusieurs variables libres, dépendra des constantes par lesquelles ses variables libres sont remplacées. Par ailleurs, la plupart des questions que l'on pose à une base de données sont des questions ouvertes, demandant un ensemble de réponses. Dès lors, pour obtenir un langage d'interrogation nous prendrons la convention suivante :
La valeur d'une formule bien formée contenant les variables libres $x_1, \ldots, x_n$ est l'ensemble des tuples $\{(a_1^{(1)}, \ldots, a_n^{(1)}), \ldots, (a_1^{(m)}, \ldots, a_n^{(m)})$ tel que $F(a_1^{(i)}, \ldots, a_n^{(i)})$ est vraie $(1 \leq i \leq m)$
Les formules correspondant à des questions ouvertes seront notées

$$\{(x_1, \ldots, x_n) \mid F(x_1, x_2, \ldots, x_n)$$

et représenteront l'ensemble des tuples $(t_1, t_2, \ldots, t_n)$ tels que la formule $F(x_1, \ldots, x_n)$ pour laquelle on a remplacé la variable x_i par $t_i (1 \leq i \leq n)$ est vraie

Les éléments du domaine.-

Pour définir les éléments du domaine d'interprétation, deux possibilités s'offrent à nous

- soit nous considérons que les éléments du domaine d'interprétation sont des tuples de relations. Les variables prendront alors leurs valeurs parmi les tuples de la base de données et on parlera de variable-tuple et de calcul relationnel à variable-tuple.
- soit nous considérons que les éléments du domaine d'interprétation sont des éléments des domaines de la base de données. Les variables prendront leurs valeurs parmi les composants des tuples de la base de données et on parlera de variable-domaine et de calcul relationnel à variable-domaine.

Nous avons défini un langage d'interrogation. Cependant, ce langage permet l'expression de formules qui paraissent sémantiquement peu significatives. En effet, si nous nous plaçons dans le cadre du calcul relationnel à variable-domaine, nous pouvons exprimer des questions de la forme $\{(x_1, x_2) / \neg P(x_1, x_2)\}$ où P est un symbole de prédicat binaire, ou des formules de la forme $\forall x \exists y\, P(x,y)$. L'évaluation de la première question nous donnera un ensemble de couples dont la plupart seront sans aucun rapport à la relation P tandis que la seconde s'évaluera presque toujours à faux. Le problème est que les variables x_1, x_2, et x prennent leurs valeurs sur tout le domaine d'interprétation. Par conséquent, il est nécessaire de restreindre le domaine sur lequel les variables prennent leurs valeurs et cela que l'on se place dans le calcul relationnel a variable-domaine ou dans le calcul relationnel à variable-tuple
A nouveau, deux possibilités s'offrent à nous.

- Soit nous nous plaçons dans une logique typée (many-sorted logic)
  Une logique typée est une logique qui admet un ensemble non vide de types $\mathcal{C}$. Chaque constante et chaque variable est associée à un type. Chaque symbole de prédicat n-aire est associé à un type $\langle T_{i1}, \ldots, T_{im} \rangle$ et T_{ij} est le type associé au jème argument de P. Chaque symbole de fonction n-aire f est associé à un type $\langle T_{k1}, \ldots, T_{km+1} \rangle$ et si $t_1, \ldots, t_n$ sont de type $T_{k1}, \ldots, T_{km}$ alors f $(t_1, \ldots, t_n)$ est de type T_{km+1}.

  Les formules bien formées de ce langage sont construites comme dans le cas non typé si ce n'est que les termes apparaissant comme arguments d'un symbole de fonction ou de prédicats doivent correspondre à leurs types respectifs.

  L'interprétation d'un langage typé consiste en n ensembles non vides d'éléments $D_1, \ldots, D_n$ respectivement associés aux types $T_1, \ldots, T_n$, ainsi que d'une assignation qui associe
  - à chaque constante d'un type donné un élément du domaine lui correspondant.
  - à chaque symbole de fonction de type $\langle T_{k1}, \ldots, T_{kp+1} \rangle$ à une fonction de $D_{k1} \ldots D_{kn}$ dans D_{k+1}.
  - à chaque symbole de prédicats de type $\langle T_{i1}, \ldots, T_{ip} \rangle$ une fonction de $D_{i1} \ldots D_{ip}$ dans $\{$vrai, faux$\}$.

  La puissance d'expression d'une logique typée est identique à celle d'une logique non typée [ende 72].

Si nous utilisons une logique typée, il faudra bien sûr que les variables soient déclarées d'un type donné.

- La deuxième possibilité consiste à utiliser des restrictions sur les formules acceptables dans le langage. On s'efforce ainsi de n'accepter que des formules qui conservent les mêmes valeurs dans deux états de la base de données qui ne diffèrent pas pour les relations qu'elles concernent ou encore des formules qui restreignent d'elles-mêmes le domaine sur lequel les variables prennent leurs valeurs. La valeur de ces formules ne dépend alors que des relations qui y interviennent. C'est ainsi que l'on peut définir les formules saines, les formules définies etc. [DEMO 81].
  Les choix qui existent entre, d'une part, le calcul relationnel à variable-tuple et le calcul relationnel à variable-domaine et, d'autre part, entre l'utilisation d'une logique typée ou d'une logique non typée, ont amené Pirotte [PIRO 78] à classifier les langages d'interrogation basé sur la logique en quatre catégories.
  Le calcul relationnel à variable-tuple sans types est à la base, par exemple des langages alpha [COOD 2] et quel [STON 76].
  Le calcul relationnel à variable-domaine sans types est représenté, par exemple, par query by example [ZLOO 77]. FQL [PIRO 77] est un représentant du calcul relationnel à variable-domaine avec types. A notre connaissance, le calcul relationnel à variable-tuple avec types n'a pas été exploitée.

Dans ce qui suit, nous nous concentrerons sur le calcul relationnel à variable-tuple ou à variable-domaine sans types qui sont les catégories les plus exploitées.

3.1.3 Le calcul relationnel à variable-tuple.

3.1.3.1 Définition du calcul relationnel à variable-tuple.

Le calcul relationnel à variable-tuple peut être défini de la manière suivante

- R(S), où R est une relation et où S est une variable-tuple, est un prédicat de rang. R(S) est vrai si S est un tuple de la relation R.

- R[i] où R est une relation et où i est un nom d'attribut de cette relation est une variable-tuple indicée. Elle représente le composant de la variable-tuple R dont le rôle est défini par l'attribut de nom i.

- Une formule atomique est définie de la manière suivante

(1) Un prédicat de rang est une formule atomique.
(2) $S \theta T$, où S et T sont des variables indicées et où θ est un opérateur de comparaison $(=, \neq, >, \geqslant, <, \leqslant)$ est une formule atomique. Cette formule exprime la comparaison entre les deux variables-tuples indicées.
(3) $S \theta$ "a", où S est une variable indicée et où "a" désigne la constante a.
(4) Il n'y a pas d'autres formules atomiques.

Une formule est définie par

(1) Toute formule atomique est une formule.
(2) Si Ψ_1 et Ψ_2 sont des formules,
$\Psi_1 \vee \Psi_2, \Psi_1 \wedge \Psi_2, \Psi_1 \rightarrow \Psi_2, \neg \Psi_1$ sont des formules.
Ψ_1 et Ψ_2 sont des sous-formules de la formule ainsi définie sauf pour $\neg \Psi_1$ où seule Ψ_1 est une sous-formule.
(3) Si Ψ est une formule et si x est une variable libre dans Ψ alors $\forall x\ (\Psi)$ est une formule.
Ψ est une sous-formule de la formule ainsi définie.
(4) Si Ψ est une formule et si x est une variable libre dans Ψ alors $\exists x\ (\Psi)$ est une formule.
Ψ est une sous-formule de la formule ainsi définie.
(5) Si Ψ est une formule, (Ψ) est une formule.
Ψ est une sous-formule de la formule ainsi définie.

Une expression du calcul relationnel à variable-tuple est une expression de la forme

$$\{(t_1, t_2, \ldots, t_n) \mid \Psi(t_1, t_2, \ldots, t_n)\}$$

où t_i est soit une variable-tuple
soit une variable-tuple indexée
Ψ est une formule qui admet $t_1, t_2, \ldots, t_n$ pour seules variables libres.

Nous imposerons cependant des restrictions sur les formules qui peuvent représenter des questions. Nous nous limiterons aux formules "safe" ou saines [ULLM 80] .

définition : Considérons une formule F où apparaissent les noms de relations $R_1, R_2, \ldots, R_k$ et les constantes $a_1, a_2, \ldots, a_p$. Le domaine de F, noté DOM (F), est défini par l'union de $a_1, a_2, \ldots, a_p$ et de l'ensemble des composants des tuples de $R_1, R_2, \ldots, R_k$.

On remarquera que, comme les relations $R_1, R_2, \ldots, R_k$ ne possèdent qu'un nombre fini de tuples, DOM (F) est un ensemble fini.

définition : Une formule F est saine (safe) si elle satisfait les trois conditions suivantes

1° Si S est un tuple tel que F(s) est vrai alors chaque composant de s est un élément de DOM(F).

2° Pour toute sous-formule de F de la forme $(\exists u)\ (F'(u))$, si s est un tuple tel que F'(s) est vrai alors chaque composant de s est un élément de DOM(F').

3° Pour toute sous-formule de F de la forme $(\forall u)(F'(u))$, s'il existe un composant de s qui n'est pas DOM(F') alors F'(s) est vrai.

Les deuxième et troisième conditions permettent d'assurer que la valeur de vérité d'une formule $\exists u\ F'(u)$ ou d'une formule $\forall u\ F'(u)$ puisse être calculée en ne considérant que les éléments du domaine de F'.

Illustrons chacune des trois conditions [ADIB 83]

exemple 1

R	
1	2
3	4

S	
4	5
3	2

Soit $F = \{(t) \mid R(t) \vee \neg S(t)\}$
$F1 = \{(t) \mid R(t) \wedge \neg S(t)\}$
$DOM(F) = DOM(F1) = \{1,2,3,4,5\}$

La formule F n'est pas saine car le tuple (3,7) satisfait "R(t) $\vee$ S(t)" mais 7 n'est pas un élément de DOM(F).
La formule F1 est saine car si un tuple SO satisfait "R(t) $\wedge \neg$ S(t)", ses composants appartiennent au domaine de "R(t)" et donc au domaine de "R(t) $\wedge \neg$ S(t)" càd à DOM(F1).

exemple 2

Soit "F(X)" une formule saine
Soit F1= "$\exists x(F(x) \wedge G(x))$"

La formule F1 satisfait la condition (2) car s'il existe x_o tel que "$F(x_o) \wedge G(x_o)$" est vrai alors $F(x_o)$ est vrai et les composants de $x_o \in DOM(F(x)) \subseteq DOM(F(x) \wedge G(x))$.

exemple 3

Soit F(x) une formule saine
Soit F1= "$\forall x(\neg F(x) \vee G(x))$"

F1 satisfait la condition (3). En effet "$\forall x(\neg F(x) \vee G(x))$" est équivalente à
- $\neg \exists x \neg (\neg F(x) \vee G(x))$
- $\neg \exists x (F(x) \wedge \neg G(x))$.

S'il existe un x_o tel que $F(x_o) \wedge \neg G(x_o)$ est vrai, alors $F(x_o)$ est vrai et les composants de $x_o \in DOM(F(x)) \subseteq DOM(F(x) \wedge G(x))$, ce qui est équivalent à la condition 3.

Cependant, si les formules saines restreignent effectivement d'elles-mêmes le domaine de leurs variables, il n'existe pas d'algorithme qui permette de déterminer si une formule est saine ou non. Dès lors, en pratique, on recourt à des classes de formules plus restreintes que les formules saines mais pour lesquelles il existe un algorithme qui permet de tester l'appartenance à la classe d'une formule donnée. Les formules à champ restreint constituent une de ces classes.
La figure 2 donne deux exemples de questions exprimées en calcul relationnel à variable-tuple utilisant la base de données de la figure 1.
[PIRO 78]

La base de données est constituée des domaines suivants :

nom, sal, dept, article, entr, étage, classe, ville

Les relations sont données par

(1) Un employé possède un nom, un salaire, un supérieur et un département

emp(empl:nom, sal, sup:nom, dept)

(2) Un département vend des articles

vend(dept, article)

(3) Une entreprise fournit des articles à un département

fournit(entr, dept, article)

(4) Un département est localisé à un étage

localisation(dept, étage)

(5) Un article appartient à une classe

type(article, classe)

(6) Une entreprise se situe dans une ville

adresse(entr, ville)

Les attributs, sauf lorsqu'ils sont indiqués explicitement, sont les noms de domaines.

figure 1.

Question 1 : Quelles sont les entreprises qui fournissent des articles 'chaussure' à un département du 2ème étage ?

$$\{(f.entr) \mid fournit(f) \wedge f.article = \text{'chaussure'} \wedge \exists l\ localisation(l) \wedge l.dept = f.dept \wedge l.étage = \text{'2'}\}$$

Question 2 : Quels sont les départements qui vendent tous les articles qu'on leur fournit ?

$$\{(v.dept) \mid vend(v) \wedge \forall f(fournit(f) \wedge f.dept = v.dept) \rightarrow \exists l(vend(l) \wedge l.dept = v.dept \wedge l.article = f.article)\}$$

figure 2.

En général, dans les langages d'interrogation, les quantificateurs existentiels sont implicites et les quantificateurs universels sont remplacés par des opérations sur des ensembles. Dans ce qui suit, nous indiquons comment construire un langage où les quantificateurs sont remplacés par des opérations sur des ensembles.

définition : une formule de rang sur une variable-tuple r est une formule qui contient uniquement des prédicats de rang concernant r et qui prend la valeur 'vrai' lorsque r appartient à une relation de la base de données ou une relation qui peut être obtenue en appliquant les relations d'union, d'intersection et de différence à des relations union-compatibles de la base de données.

Considérons $P(r)$ une formule de rang sur la variable r.
$Q(r)$ et $R(r)$ deux formules qui ne contiennent pas de prédicats de rang concernant r et qui admettent r comme variable libre.

Convenons que $\emptyset$ représente l'ensemble vide.
On peut définir les équivalences suivantes :

- $\exists r(P(r) \wedge Q(r))$ est équivalent à $\{r \mid P(r) \wedge Q(r)\} \neq \emptyset$
- $\forall r(P(r) \rightarrow Q(r))$ est équivalent à $\{r \mid P(r)\} = \{r \mid P(r) \wedge Q(r)\}$
- $\forall r(P(r) \wedge Q(r) \rightarrow R(r))$ est équivalent à
$$\{r \mid P(r) \wedge Q(r)\} \subseteq \{r \mid P(r) \wedge R(r)\}.$$

Par conséquent, un langage d'interrogation peut être construit sur base de ces transformations. Pour ce faire, il suffit de supprimer les points (3) et (4) de la définition des formules et définir les notions suivantes.

Une expression d'ensemble est définie par

(1) $\emptyset$ est une expression d'ensemble qui représente l'ensemble vide

(2) Si Ψ est une formule qui admet x comme variable libre $\{x \mid \Psi(x)\}$ où x est une variable-tuple est une expression d'ensemble. Elle représente l'ensemble des tuples x tels que $\Psi(x)$ est vrai. x est lié à l'expression $\{x \mid \Psi(x)\}$ de la même manière qu'une variable quantifiée est liée à son quantificateur.

(3) Si E_1 et E_2 sont des expressions d'ensemble, alors
$$E_1 \theta' E_2$$ est une formule où

θ' est un opérateur de comparaison d'ensemble pris dans
$\{=, \subseteq, \subset, \nsubseteq, \not\subset, \neq\}$

Les questions de la figure 2 sont exprimées dans ce langage à la figure 3.
Enfin, il est possible d'admettre des expressions d'ensemble $\{x \mid \Psi(x)\}$ où x n'est plus une variable-tuple mais une variable indicée.
Le langage Quel [STON 76] est un langage basé sur le calcul relationnel à variable-tuple où les quantificateurs existentiels sont implicites et où les quantificateurs universels sont remplacés par des opérations sur ensembles avec des expressions d'ensembles sur variable indicée.

3.1.3.2 Mise en oeuvre du calcul relationnel à variable-tuple.

Dans ce qui suit, nous allons montrer comment Quel [STON 76] , le langage d'interrogation d'Ingrès, peut être considéré comme une mise en oeuvre du calcul relationnel à variable-tuple sans types. L'objectif n'est pas de présenter Quel mais d'illustrer la partie théorique qui précède.

La forme traditionnelle des questions en Quel est la suivante

$$\{(t_{i_1}[A_{i_1}], \ldots, t_{i_n}[A_{i_n}]) \mid \exists\, t_1, t_2, \ldots, t_k \; R1(t1) \wedge \ldots \wedge R_k(t_k) \wedge \Psi(t_1, t_2, \ldots, t_k)\}$$

où - $\Psi(t_1, t_2, \ldots, t_k)$ est une formule sans quantificateurs et sans prédicats de rang, qui admet $t_1, t_2, \ldots, t_k$ comme variables libres.
- $1 \leqslant i_j \leqslant k$ pour $1 \leqslant j \leqslant n$.

Cette forme donne lieu à la question Quel

```
range of t1 is R1
   :
   :
range of tk is Rk
retrieve (ti1 [ Ai1 ], ... , tin [ Ain ] )
        where  Ψ (t1, ... , tk) .
```

exemple : la question 1 énoncée précédemment devient

```
range  of f is fournit
range of l is localisation
```

Question 1 : Quelles sont les entreprises qui fournissent des articles 'chaussure' à un département du deuxième étage ?

$$\{(f.entr) \mid fournit(f) \wedge f.article = \text{'chaussure'} \wedge \{ l \mid localisation(l) \wedge f.dept = l.dept \wedge l.étage = \text{'2'} \} \neq \emptyset \}.$$

Question 2 : Quels sont les départements qui vendent tous les articles qu'on leur fournit ?

$$\{(v.dept) \mid vend(v) \wedge \{ f \mid fournit(f) \wedge f.dept = v.dept \} \subseteq \{ f \mid fournit(f) \wedge \{ l \mid vend(l) \wedge v.dept = l.dept \wedge l.article = f.article \} \neq \emptyset \} \}.$$

Si l'on permet des expressions d'ensemble avec variable-indicée, cette dernière expression peut être simplifiée en

$$\{(v.dept) \mid vend(v) \wedge \{f.article \mid fournit(f) \wedge f.dept = v.dept \subseteq \{f.article \mid vend(l) \wedge l.dept = v.dept\} \}.$$

figure 3.

```
retrieve (f.entr)
    where f.article = 'chaussure' and
          f.dept = l.dept and
          l.étage = '2'
```

Nous pouvons nous apercevoir que les quantificateurs existentiels sont implicites. De même, on remarquera que les questions de cette forme sont nécessairement saines. Cependant, cette forme de questions n'est pas suffisamment générale. Elle ne permet pas l'expression de formules quantifiées universellement. Les quantificateurs universels n'apparaitront jamais en Quel et seront remplacés par des opérateurs de manipulation d'ensembles. Nous allons présenter les trois opérations nécessaires "retrieve into", "append to" et "delete".

NB. Nous faisons ici l'hypothèse que Quel effectue effectivement des opérations de manipulation d'ensembles. En réalité, Quel ne supprime pas les doubles mais l'opérateur 'Sort' permet d'obtenir le résultat escompté.

opération "retrieve into"

L'opération "retrieve into" permet de créer un ensemble de tuples à partir des relations existantes.
Soit S une nouvelle relation (i.e.: non encore présente dans la base de données).
La forme de l'opération est la suivante :

range of t_1 is R_1
⋮
range of t_k is R_k
retrieve into $S(A_1 = W_1, \ldots, A_n = W_n)$
where $\Psi(t_1, \ldots, t_k)$

où - $\Psi(t_1, \ldots, t_k)$ est une formule sans quantificateurs et prédicats de rang avec $t_1, \ldots, t_k$ comme variables libres
- $W_i (1 \leqslant i \leqslant n)$ peut se restreindre, dans le cadre qui nous préoccupe, à des expressions construites à partir de constantes, de variables-tuple indicées et d'opérateurs arithmétiques (+,x,-,:).

L'effet de cette opération est de trouver tous les ensembles de tuples $\{ t_1, \ldots, t_k \}$ tels que t_j est un tuple de la relation R_j $(1 \leqslant j \leqslant k)$ et tels que $\Psi(t_1, \ldots, t_k)$ est vrai, et de créer

pour chacun de ces ensembles, un nouveau tuple pour la nouvelle relation S dont la valeur, pour le composant dont le nom d'attribut associé est A_i, est W_i $(1 \leqslant i \leqslant n)$

opération "append to"

Cette opération permet d'ajouter un ensemble de tuples à une relation.

Sa forme est la suivante

```
range of t1 is R1
   :
range of tk is Rk
append into S(A1 = W1, ... , An = Wn)
      where Ψ (t1, ... ,tk)
```

$\Psi(t_1, \ldots, t_k)$ et W_i $(1 \leqslant i \leqslant n)$ sont définis comme précédemment.

L'effet de cette opération est de trouver tous les ensembles de tuples $\{ t_1, \ldots, t_k \}$ tels que t_j appartient à la relation R_j $(1 \leqslant i \leqslant k)$ et tels que $\Psi (t_1, \ldots, t_k)$ est vrai et d'ajouter, pour chacun de ces ensembles, un nouveau tuple à la relation dont la valeur est donnée pour le composant dont l'attribut associé a le nom A_i, par W_i $(1 \leqslant i \leqslant n)$.

opération "delete"

Cette opération permet de supprimer un ensemble de tuples d'une relation.

Sa forme peut être définie par

```
range of t is R
delete t
      where Ψ (t)
```

où $\Psi (t)$ est défini comme précédemment.

L'effet de cette opération est de supprimer tous les tuples t, tels que $\Psi (t)$ est vrai, de la relation R.

La figure 4 illustre l'utilisation de ces opérations pour la deuxième question.

On a montré que Quel, muni de ces opérations sur ensembles, avait la même puissance d'expression que l'algèbre relationnelle. [ULLM 80]

Question 2 : Quels sont les départements qui vendent tous les articles qu'on leur fournit ?

```
range of f is fournit
retrieve into art(dept = f.dept,art = f.article)
range of a is art
range of v is vend
delete a
     where a.dept = v.dept and
           a.art = v.art
retrieve (f.dept)
     where
         f.dept ≠ a.dept
```

figure 4

3.1.4 Le calcul relationnel à variable-domaine.

3.1.4.1 Définition du calcul relationnel à variable-domaine.

Le calcul relationnel à variable-domaine peut être défini de la manière suivante :

- un terme est soit une constante soit une variable-domaine,

- une formule atomique est définie par
  (1) $R(t_1, \ldots, t_n)$ où R désigne le nom d'une relation n-aire et où $t_1, t_2, \ldots, t_n$ sont des termes.
  (2) $t_1 \ \theta \ t_2$ où t_1 et t_2 sont des termes et θ représente un des opérateurs arithmétiques de comparaison.
  (3) Il n'y a pas d'autres formules atomiques.

- une formule est définie par
  (1) les formules atomiques sont des formules,
  (2) Si F1 et F2 sont des formules, alors
  $F1 \wedge F2$, $F1 \vee F2$, $F1 \rightarrow F2$ et $\neg F2$ sont des formules.
  F1 et F2 sont des sous-formules de la formule ainsi définie sauf pour $\neg F2$ où seule F2 est une sous-formule.
  (3) Si F1 est une formule et x est une variable libre dans F1, alors $\forall x F1$ et $\exists x F1$ sont des formules.
  F1 est une sous-formule de la formule ainsi définie.
  (4) Si F1 est une formule, alors (F1) est une formule.
  F1 est une sous-formule de la formule ainsi définie.
  (5) Il n'y a pas d'autres formules.

Une expression du calcul relationnel à variable-domaine est une expression de la forme

$$\{(x_1, \ldots, x_n) \mid F(x_1, \ldots, x_n)\}$$

où $x_1, \ldots, x_n$ sont les seules variables libres de F.

De la même manière que précédemment, nous dirons qu'une expression du calcul relationnel à variable-domaine $\{(x_1, x_2, \ldots, x_n) \mid \Psi(x_1, x_2, \ldots, x_n)\}$ est saine si
(1) Si $\Psi(x_1, x_2, \ldots, x_n)$ est vrai, alors x_i $(1 \leq i \leq n) \in Dom(\Psi)$.
(2) Si $\exists u W(u)$ est une sous-formule de Ψ, alors si W(s) est vrai, alors s est dans le Dom(W).

(3) Si $\forall$ uW(u) est une sous-formule de ψ , alors si W(s) est faux, alors s est dans le Dom(W).

Comme dans le calcul relationnel à variable-tuple, nous pouvons supprimer les quantificateurs et les remplacer par des opérations sur ensembles. Cependant, il faudra s'assurer que les expressions d'ensemble obtenues restent saines. Une expression d'ensemble $\{(x_1, x_2, \ldots, x_n) \mid F(x_1, x_2, \ldots, x_n)\}$ est saine si $F(x_1, x_2, \ldots, x_n)$ est saine.
On peut définir les équivalences suivantes :

- $\exists x\ F(x)$ est équivalent à $\{x \mid F(x)\} \neq \emptyset$
- Si la formule est '$\forall x\ F(x) \rightarrow G(x)$' et que la formule 'F(x)' détermine le domaine sur lequel x prend ses valeurs, elle est équivalente à

$$\{x \mid F(x)\} = \{x \mid F(x) \wedge G(x)\} .$$

La figure 5 illustre les deux questions pour ce dernier langage. Le langage Query by example peut être considéré comme une mise en oeuvre du calcul relationnel à variable-domaine où les quantificateurs existentiels sont implicites et où les quantificateurs universels sont remplacés par des opérations sur ensembles.

3.1.4.2 Mise en oeuvre du calcul relationnel à variable-domaine.

Comme précédemment, l'objectif est d'illustrer les propos théoriques. Les questions du calcul relationnel à variable-domaine de la forme

$$\{(a_1, .., a_n) \mid \exists b_1\ \exists b_2\ \exists b_m (R_1(c_{11}, .., c_{1k1}) \wedge \ldots \wedge R_p(c_{p1}, .., c_{pkp})\}$$

où chaque c_{ij} est soit un a_t $(1 \leqslant t \leqslant n)$
soit un b_t $(1 \leqslant t \leqslant m)$
soit une constante.

où chaque a_t et b_t apparaissent au moins une fois dans les c_{ij}.

peuvent être transformées en Query by example de la manière suivante :

(1) faire apparaître pour chaque relation R_1, R_2, .., R_p un squelette de table lui correspondant

(2) créer, pour chaque formule atomique $R_i(c_{i_1}, .., c_{i_{ki}})$, une ligne dans le squelette correspondant

- Si c_{ij} est une constante, alors elle peut être placée dans la colonne lui correspondant

Question 1 : Quelles sont les entreprises qui fournissent des chaussures à un département du second étage ?

fournit	entr	dept	article
	P.X	départ	chaussure

localisation	dept	étage
	départ	2

X, départ sont des variables.

figure 6.

Question : Quels sont les départements qui vendent tous les articles de la classe A et seulement ceux-là ?

Sous forme de calcul relationnel à variable-domaine sans quantificateurs :

$\{d \mid \{a \mid type(article : a, classe : 'A')\} = \{a \mid vend(dept : d, article : a)\}\}$

La question en Q.B.E. est donnée par

vend	dept	art
	P.$\underline{\underline{D}}$	all.$\underline{a}$

type	article	classe
	all.$\underline{a}$	A

$\underline{a}$, $\underline{\underline{D}}$ représentent des variables.

La table de droite, prise isolément nous assure que "all.a" est l'ensemble de tous les articles de la classe A.
La table de gauche nous donne les départements qui vendent tous les articles représentés par 'all.a'. Le département est un facteur de groupement et c'est la raison pour laquelle il est souligné deux fois.

figure 7.

Question : Quels sont les départements qui vendent au moins tous les articles de la classe A ?

Sous forme de calcul relationnel à variable-domaine sans quantificateurs :

$$\{d \mid \{a \mid type(article : a, classe : 'A')\} = \{a \mid vend(dept : d, article = d)\}\} .$$

vend	dept	art
	$\underline{D}$	$\left[\underline{all.a}\right]_{*}$

type	article	classe
	$\underline{all.a}$	A

où $\left[\underline{all.a}\right]_{*}$ représente un ensemble qui contient l'ensemble "all.a" plus éventuellement d'autres éléments.

figure 9.

- Si c_{ij} est une variable qui n'apparait qu'une fois, alors sa colonne peut rester blanche. Cette variable n'exprime en effet, aucune condition sur les éléments recherchés.
- Sinon la variable correspondant à c_{ij} est placée dans la colonne lui correspondant.

(3) dans la plupart des questions, les $a_i (1 \leqslant i \leqslant n)$ seront les arguments d'une et seule formule atomique $R_i(c_{i_1}, .., c_{i_{ki}})$ ce qui signifie que l'on recherche des tuples dont les composants appartiennent à une relation. Dans ce cas, il suffit de préfixer les variables $a_i (1 \leqslant i \leqslant n)$ par l'opérateur P pour obtenir le résultat souhaité. L'opérateur P est l'opérateur d'impression. Dans le cas où toutes les variables $a_i (1 \leqslant i \leqslant n)$ ne sont pas les arguments d'une formule atomique, il sera nécessaire de créer un nouveau squelette de table qui contiendra toutes les variables a_i. Il suffit alors de placer $P.a_1$ $P.a_2$ ... $P.a_n$.

La figure 6 illustre la première question en Query by example. Query by example dispose également d'opérations sur ensembles équivalentes à celles que nous avons vues en Quel mais adaptées au calcul relationnel à variable-domaine. De plus, Query by example dispose d'une fonction "all." qui permet de construire des ensembles qui pourront ensuite être comparés. (En fait, elle construit des "ensembles" avec doubles mais il est possible de supprimer ces doubles). Les figures 7 et 8 illustrent l'utilisation de cette fonction.

On remarquera que les questions sont toujours saines, le domaine sur lequel une variable prend ses valeurs étant l'ensemble des composants des tuples de la relation dont le nom d'attribut est identique au nom de la colonne du squelette. De plus, on a montré que Query by example était aussi puissant que l'algèbre relationnelle [ULLM 80].

3.1.5 Logique typée versus logique non typée : quelques réflexions.

Les langages d'interrogation que nous avons considérés jusqu'ici sont basés sur une logique non typée. Il est également possible, comme nous l'avons mentionné, de baser un langage d'interrogation

sur une logique typée et nous pensons d'ailleurs que cette solution présente certains avantages. L'utilisation de types est possible à la fois pour le calcul relationnel à variable-tuple et le calcul relationnel à variable-domaine. Cependant, elle est plus compliquée dans le cadre du calcul relationnel à variable-tuple car, dans ce cas, il est nécessaire de définir des types non seulement por les variables-tuples mais aussi pour les variables-indicées [PIRO 78]. Dans ce qui suit, nous nous placerons donc dans le contexte du calcul relationnel à variable-domaine.

De manière générale, l'utilisation ou non de types dans un langage est un compromis entre la souplesse d'expresssion du langage et son efficacité d'implémentation. Dans le cas des langages d'interrogation, elle permet de définir le domaine sur lequel les variables prennent leurs valeurs et il n'est donc plus nécessaire de déterminer, lors de l'évaluation d'une question, le domaine de ces variables. Par ailleurs, l'utilisation de types permet d'effectuer un certain nombre de contrôles syntaxiques et de rejeter des questions dépourvues de sens. On peut aussi s'assurer que les deux arguments d'un opérateur de comparaison sont de types identiques ou que les arguments d'un prédicat appartiennent aux types leur correspondant dans la définition du prédicat. La nécessité d'une logique typée s'est également fait sentir pour rejeter des questions dépourvues de sens lors de l'interrogation d'une base de données en langage naturel. [DALH 82].

Cependant, l'utilisation d'une logique typée peut figer l'évolution d'une application ou rendre impossible l'expression de certaines questions. Il ne faut cependant pas perdre de vue qu'un des objectifs d'une base de données est la modélisation du monde réel ou du moins d'une réalité et que par conséquent, une base de données doit être conçue en considérant la notion d'extension et d'évolution. Dès lors, une modélisation adéquate devrait permettre de remédier aux problèmes mentionnés dans une large mesure. Cela nous amène à considérer le rôle que joue la notion de domaine dans le modèle relationnel.

En effet, bien que la notion de domaine soit présente dans le modèle relationnel, elle a été relativement peu exploitée. C'est ainsi que dans une base de données relationnelles, les domaines sont souvent des classes syntaxiques comme les entiers, les strings de 20 caractères, etc... Cependant, plutôt que d'utiliser les domaines comme des concepts tournés vers l'implémentation, il est également possible de les associer à des concepts sémantiquement significatifs eu égard à l'application. Les domaines représenteront donc des employés, des

entreprises, des articles, etc... Le fait qu'un domaine apparaisse à plusieurs endroits dans un schéma relationnel exprime de manière explicite une information sémantique qui ne peut être représentée par des attributs ou des relations [LACR 80]. A chaque domaine sera associé un type et si la modélisation est effectuée soigneusement, il devrait être possible d'exprimer les questions sémantiquement significatives et de rejeter les questions dépourvues de sens.

3.2 Mise en oeuvre des questions.

Nous venons d'indiquer comment la logique du premier ordre peut être modifiée pour définir un langage d'interrogation. Nous obtenons alors le calcul relationnel à variable-domaine et à variable-tuple. Ces langages permettent à un utilisateur d'exprimer sa question sans avoir à penser à la manière dont il va accéder aux données. De plus, on considère généralement que ces langages sont "de plus haut niveau" que l'algèbre relationnelle car contrairement à cette dernière, la question ne donne pas un algorithme pour calculer les réponses.

Cependant, pour une mise en oeuvre efficace de ces langages, il sera nécessaire de définir des techniques d'optimisation de questions. Différentes formes d'optimisation peuvent être effectuées. On distingue essentiellement deux formes complémentaires :

- la première consiste à transformer la question initiale en une question optimisée qui lui est équivalente mais dont le coût d'évaluation est moindre.
- la seconde consiste à déterminer, à partir d'une question, un certain nombre de séquences d'opérations élémentaires sur la base de données, chacune d'entre elles produisant le résultat escompté, puis d'évaluer le coût de chacune de ces séquences et à retenir la meilleure.

La logique est à la base d'un certain nombre de transformations de la première forme. Ces transformations elles-mêmes peuvent être scindées en deux catégories :

- les transformations qui préservent l'équivalence entre la question initiale et la question optimisée quelle que soit la base de données. Ces transformations sont parfois appelées 'optimisations syntaxiques'.
- les transformations qui préservent l'équivalence entre la question initiale et la question optimisée uniquement dans le cadre d'une application. Ces transformations sont parfois appelées 'optimisations sémantiques'.

3.2.1 Optimisations syntaxiques.

Les optimisations syntaxiques tirent profit d'un certain nombre d'observations. Nous allons en présenter brièvement quelques-unes. Le lecteur intéressé peut se reporter à [JARK 84] où il trouvera des références appropriées.

Une question est une formule logique. Dès lors, cette formule peut contenir des sous-expressions identiques et peut être simplifiée.
De même, nous pouvons tenir compte de la sémantique du prédicat d'égalité '=' pour propager à travers la question des constantes en lieu et place de variables.
Dans le même ordre d'idées, la sémantique des opérateurs de comparaison peut être utilisée pour en déduire qu'une question est incohérente dans certains cas.
Des approches plus globales [ULLM 80] se restreignent à des classes de questions (par ex : les questions conjonctives)pour ensuite à partir d'une question en déduire une autre, équivalente mais qui peut être montrée minimale.

Exemple : la question

$$\{(a_3,a_4) \mid \exists b_3 \exists b_4 \exists b_5 \exists b_6 \mid R(b_3,b_5) \wedge R(a_3,b_6) \wedge R(b_4,c) \wedge S(b_4,a_4)\}$$

est équivalente à la question

$$\{(a_3,a_4) \mid \exists b_4 \exists b_6 \; R(a_3,b_3) \wedge R(b_4,c) \wedge S(b_4,a_4)\}$$

qui peut être montrée minimale.

De même, ces techniques de simplification peuvent être étendues pour prendre en compte des informations sémantiques concernant les dépendances fonctionnelles et multivaluées. En ce faisant, on se dirige vers les optimisations sémantiques.

3.2.2 Optimisations sémantiques.

Nous avons mentionné que les optimisations sémantiques ne conservaient l'équivalence entre la question initiale et la question optimisée que dans le cadre d'une application donnée. Ces techniques utilisent, en effet, les contraintes d'intégrité pour générer la forme optimisée. Ces techniques ont été introduites indépendamment par Hammer et King [HAMM 80] [KING 81]
Elles sont pour l'optimisation des questions ce que sont les stratégies globales de résolution [KOWA 79] pour la programmation logique. Elles consistent essentiellement en la suppression d'informations sémantiquement redondantes, l'insertion d'informations sémantiquement redon-

dantes et la détection de questions incohérentes. Nous allons illustrer ces dernières techniques.

Suppression d'informations sémantiquement redondantes.

Certaines parties d'une question peuvent être logiquement impliquées par le reste de la question et les contraintes d'intégrité. Par conséquent, elles peuvent être supprimées sans changer la sémantique associée à la question.

Exemple : Soit la relation

personne (nom-p, sal-p, fonction-p)

exprimant le nom, le salaire et la fonction d'une personne. La relation dispose d'un index portant sur le nom 'nom-p' et un index portant sur 'fonction-p'.

Considérons la contrainte

'Tous les directeurs gagnent plus de 150.000 F'

La question

'Quelles sont les personnes qui sont directeurs et qui gagnent plus de 120.000 F?'

peut être optimisée en

'Quelles sont les personnes qui ont comme fonction directeur ?'

La suppression d'informations redondantes est particulièrement adéquate lorsqu'elle permet de supprimer des jointures entre relations.

Insertion d'informations sémantiquement redondantes.

Nous pouvons également envisager d'ajouter à la question des conditions qui, bien que sémantiquement redondantes (la question augmentée est équivalente à la question initiale), permettent d'augmenter la performance lors de l'évaluation de la question. On pourra par exemple, introduire un index, réduire les éléments concernés par une jointure et ainsi de suite.

Exemple : La question considérée est

'Quelles sont les personnes qui gagnent plus de 200.000 F?'

La contrainte considérée est

'Toute personne qui gagne plus de 150.000 F est directeur'

La question optimisée est

'Quels sont les directeurs qui gagnent plus de 200.000 F?'

La question optimisée a fait apparaitre un index pour l'accès à la relation 'personne'.

Détection de questions incohérentes.

Dans certains cas, une question peut être incohérente par rapport à l'ensemble des contraintes d'intégrité. Dès lors, la question ne devra pas être évaluée puisque par définition elle ne peut être satisfaite par aucune valeur.

Exemple : La question

'Quelles sont les personnes qui gagnent plus de 160.000 F et qui ne sont pas directeurs ?'

est incohérente par rapport à la contrainte

'Toute personne qui gagne plus de 150.000 F est directeur'.

Par conséquent, elle ne devra pas être évaluée.

Cette dernière catégorie d'optimisations a été largement exploitée par Minker et Reiter [MINK 78] [REIT 78a] pour la détection de questions incohérentes à partir d'informations concernant le domaine des variables.

A partir d'une question et d'un ensemble de contraintes, il peut être possible de générer un grand nombre de formules équivalentes. Toutes ne sont pas utiles, bien au contraire. Cependant, il est impensable de générer toutes ces formules, de les évaluer vis-à-vis d'un critère de sélection et d'en déduire la meilleure. Il faudra s'efforcer autant que possible de réduire l'arbre de recherche des formules équivalentes et de déterminer des heuristiques qui pourront nous permettre de nous guider dans le parcours de l'arbre. Le problème réside essentiellement à trouver ces heuristiques. Nous en avons présentées quelques-unes. D'autres peuvent être trouvées dans [KING 81]. A notre connaissance, les travaux réalisés jusqu'à ce jour sont très limités et doivent être considérés comme une première étape. On se restreint à des classes particulières de questions et les heuristiques doivent être développées.

La programmation en logique est un des outils adéquats pour la mise en oeuvre de ces optimisations.

CHAPITRE 4. CONTRAINTES D'INTEGRITE.

4.0 Introduction.

La partie intégrité constituait le troisième volet de notre présentation du modèle relationnel. Ce chapitre sera donc consacré à l'apport de la logique des prédicats du premier ordre aux contraintes d'intégrité. Tout comme les langages d'interrogation de base de données relationnelle, la mise en oeuvre des contraintes d'intégrité demande deux composants :

- un langage d'expression des contraintes.
- un mécanisme de mise en oeuvre qui assure la cohérence de la base de données par rapport aux contraintes définies.

Ce chapitre traitera des deux composants. Dans un premier temps, nous étudierons les moyens d'expression des contraintes et nous nous attarderons sur les contraintes de transition.

Ensuite, nous aborderons l'apport de la logique des prédicats du premier ordre (par l'intermédiaire de la formalisation des bases de données selon la vue "théorie du modèle") pour la mise en oeuvre des contraintes. Dans ce cadre, nous présenterons une méthode de simplification de contraintes d'intégrité. Cette méthode sera située dans le cadre plus général des mécanismes de simplification et nous montrerons ses qualités, faiblesses et les extensions possibles.

4.1 Représentation des contraintes d'intégrité.

Les contraintes d'intégrité, traitées dans les chapitres précédents, étaient des contraintes d'intégrité d'état. Ces contraintes d'intégrité pouvaient être interpretées comme des questions fermées qui réclament comme réponse "oui" lorsqu'elles sont évaluées. Puisque les questions fermées ne sont en fait qu'un cas particulier des questions ouvertes, il est naturel de dire que les langages d'interrogation constituent un formalisme adéquat pour ce type de contraintes. Les développements que nous avons effectués au chapitre précédent restent donc valables pour les contraintes d'intégrité d'état.
Cependant, si les langages d'interrogation semblent adéquats pour exprimer les contraintes d'intégrité d'état, il n'est pas évident qu'ils puissent exprimer les contraintes de transition. En effet, par définition, les contraintes de transition sont des contraintes qui imposent des conditions sur les transitions entre différents états de base de

données. La sémantique des langages d'interrogation, définie dans le chapitre précédent, est basée sur la notion d'interprétation qui ne concerne qu'un seul état de la base de données. Par conséquent, on voit mal comment ces langages pourraient permettre l'expression des contraintes de transition.
Certains systèmes (Sequel [ASTR 76] par exemple) permettent l'expression des contraintes de transition en préfixant les variables à l'aide des mots réservés "old" et "new". Ces préfixes permettront respectivement de désigner des valeurs dans l'état courant de la base de données et ces mêmes valeurs dans l'état de la base de données après une mise à jour. Ces préfixes, avec la possibilité d'exprimer quand la contrainte d'intégrité doit être vérifiée, permettent d'exprimer les contraintes de transition.
Cette notation nous semble cependant malheureuse car elle est contraire à la notion d'interprétation et la sémantique définie pour les langages d'interrogation n'a plus guère de sens si l'on permet l'utilisation de ces préfixes. Une autre possibilité est décrite dans [NICO 78]. Elle consiste à utiliser des relations spécialement définies pour ces contraintes d'intégrité, appelées relations d'action.
Nous allons nous attarder quelque peu sur cette notation.
Il existe en effet trois formats de relation d'action, chacune correspondant à un type particulier de mise à jour (modification, insertion, suppression).

Considérons dans un premier temps, les contraintes de modification.
Soit $R(x_1, x_2, \ldots, x_n)$ une relation n-aire et nous désirons exprimer une contrainte de transition en cas de modification de R.
Pour ce faire, il est nécessaire de définir une relation

$$\text{modification-}R(x_1, x_2, \ldots x_n, x_1', \ldots, x_n').$$

La sémantique de cette relation est la suivante : si l'on tente une modification de la relation R, l'extension de la relation contient le seul tuple $\langle a_1, a_2, \ldots, a_n, a_1', \ldots, a_n' \rangle$
où $\langle a_1, a_2, \ldots, a_n \rangle$ représente le tuple qui va être modifié dans R
$\langle a_1', a_2', \ldots, a_n' \rangle$ représente la nouvelle valeur de ce tuple.
Dans tous les autres cas, l'extension de la relation est vide.

La contrainte de transition va exprimer certaines conditions sur les composants d'un tuple dans la relation "modification-R".
C'est ainsi qu'une contrainte définira un ensemble $\bigvee$ de composants qui

varient pendant que d'autres restent constants, les composants restants étant sans intérêt pour la contrainte.

exemple : considérons la relation EMP (num-E, ville-E, sal-E) qui donne la ville et le salaire d'un employé, la contrainte "le salaire d'un employé ne peut que croître", $\langle a_1, a_2, a_3 \rangle$ le tuple qui va être modifié et $\langle a_1', a_2', a_3' \rangle$ le tuple qui va le remplacer. La contrainte ne risque d'être falsifiée que si

$$a_1 = a_1' \text{ et } a_3 \neq a_3'$$

la valeur de a_2' par rapport à a_2 n'ayant aucun intérêt pour la contrainte.

Par conséquent la contrainte devra spécifier ces conditions pour correspondre à sa sémantique. C'est ainsi que la contrainte correspondant à l'exemple précédent sera exprimée (en calcul relationnel à variable-domaine) par $\forall x_1 \forall x_2 \forall x_3 \forall x_1' \forall x_2' \forall x_3'$ modification-emp$(x_1, x_2, x_3, x_1', x_2', x_3') \wedge (x_1 = x_1') \wedge (x_3 \neq x_3') \rightarrow (x_3' > x_3)$.

La figure 4. 1 donne la contrainte de transition bien connue sur l'état civil.

Tournons-nous maintenant vers les contraintes de transition en insertion (les contraintes de transition en suppression peuvent être définies par analogie).
Pour exprimer une contrainte de transition lors de l'insertion d'un tuple dans une relation R, il est nécessaire de créer une relation d'action

insertion-R$(x_1, x_2, \ldots, x_n)$ où n est l'arité de R.

Sa sémantique est la suivante : insertion-R$(x_1, x_2, \ldots, x_n)$a une extension qui contient le tuple $\langle a_1, a_2, \ldots, a_n \rangle$ lorsque l'on tente d'insérer $\langle a_1, a_2, \ldots, a_n \rangle$ dans l'extension de R. Dans tous les autres cas, son extension est vide.

exemple : considérons les relations

membre-tennis(<u>personne, club</u>) qui exprime l'appartenance d'une personne à un club de tennis

entreprise(<u>personne, entr</u>) qui exprime l'entreprise dans laquelle travaille une personne

et la contrainte

"Pour devenir membre du club "Rail-Tennis", il est nécessaire de travailler à la SNCB".

Cette contrainte s'exprimera de la manière suivante :

Soit la relation

état civil(personne, état)

qui donne, pour une personne, son état civil

Soit la relation

trans-état(ancien, nouveau)

qui donne la transition d'un état civil vers un autre.

La contrainte peut s'exprimer par

$$\forall x\ \forall x'\ \forall y\ \forall y'\ (\text{modification-état-civil}(x,y,x',y') \wedge (x=x') \wedge (y=y') \longrightarrow \text{trans-état}(y,y')$$

L'extension de la relation "trans-état (ancien, nouveau)" est donnée par

trans-état	ancien	nouveau
	célibataire	marié
	marié	divorcé
	marié	veuf
	divorcé	marié
	veuf	marié

figure 4.1.

$\forall x$ insertion-membre-tennis(x,"Rail-Tennis") $\rightarrow$ entreprise (x,"SNCB")

Cette contrainte est essentiellement distincte de la contrainte d'état
$\forall x$ membre-tennis(x,"Rail-Tennis") $\rightarrow$ entreprise (x,"SNCB").
Dans ce dernier cas, en effet, une personne ne pourra être membre du club"Rail-Tennis" que tant qu'elle travaille à la SNCB.

Enfin, il peut exister des contraintes qui mettent en cause des modifications à plusieurs relations. Ces contraintes exigent que les modifications correspondantes soient traitées comme des transactions.

exemple : Soit les relations grade(militaire,type-grade)
salaire(militaire,sal)
exprimant respectivement le grade d'un militaire et son salaire. La contrainte "Tout changement de grade est accompagné d'un changement de salaire" est exprimée par

$$\text{modification-grade}(x,z,x,z') \wedge (z \neq z') \rightarrow \text{modification-salaire}(x,y,x,y') \wedge (y \neq y').$$

Il devrait être clair que toute modification de grade suivie d'une modification de salaire doit être considérée comme une séquence indivisible eu égard à cette contrainte.

Que conclure de tout ceci ? D'abord, les contraintes d'état ne présentent aucune difficulté en ce qui concerne leurs expressions et les langages d'interrogation peuvent être aisément utilisés pour les exprimer. Les contraintes de transition présentent, quant à elles, des difficultés compte tenu de la manière dont la sémantique est définie dans les langages d'interrogation. Nous avons mentionné une première manière de les traiter et nous avons montré pourquoi elle ne nous semblait pas adéquate. Nous avons ensuite présenté une seconde manière de les exprimer. Celle-ci peut sembler quelque peu artificielle puisqu'elle nécessite l'introduction de nouvelles relations. Cependant, sa principale qualité réside dans le fait qu'elle n'altère pas la sémantique des langages d'interrogation.

relations :

- vend(département, article)
  exprime qu'un département vend un article
- fournit(entreprise, département, article)
  exprime qu'une entreprise fournit un article à un département.
- subord(subordonné, supérieur)
  exprime la relation entre un employé et son supérieur.

contrainte 1 :un département ne peut vendre un article que si une entreprise le lui fournit.

$\forall x \forall y \exists z \ (\neg \text{vend}(x,y) \vee \text{fournit}(z,x,y))$.

contrainte 2 :Tout employé, qui est le subordonné d'un employé qui est lui-même le subordonné d'un autre employé, est subordonné de ce dernier.

$\forall x \forall y \forall z \ (\neg \text{subord}(x,z) \vee \neg \text{subord}(z,y) \vee \text{subord}(x,y))$.

figure 4.2.

4.2 Mise en oeuvre des contraintes d'intégrité.

4.2.1 Cadre de référence.

Si la plupart des systèmes de base de données relationnelle permettent l'expression des contraintes d'intégrité, peu d'entre eux offrent les mécanismes nécessaires à la mise en oeuvre de ces contraintes ou bien se restreignent à des classes réduites de contraintes. Cood [COOD 82]reconnaissait d'ailleurs que la partie intégrité constituait une partie faible des systèmes de gestion de base de données relationnelle.

La raison est un problème de performances. Il est, en effet, impensable de vérifier toutes les contraintes d'intégrité lors d'une mise à jour. De plus, même après avoir déterminé l'ensemble des contraintes qui peuvent être falsifiées par une mise à jour, la vérification d'une contrainte, comme d'ailleurs l'évaluation d'une question, est un processus coûteux en temps. Par conséquent, il est nécessaire de réduire ce coût si nous désirons mettre en oeuvre efficacement le processus de vérification des contraintes.

Un certain nombre de considérations concernant les contraintes d'intégrité et le processus de vérification ont mis en évidence certaines voies de recherches qui peuvent contribuer à l'amélioration du processus de vérification des contraintes.

Nous en avons retenu cinq, tout en sachant fort bien que cette liste n'est en rien limitative.

1.- Une contrainte d'intégrité est une question fermée.
    Par conséquent, il est clair que les techniques d'optimisation mises en oeuvre pour les langages d'interrogation pourront être utilisées pour la vérification des contraintes d'intégrité.
2.- Une contrainte d'intégrité ne risque pas d'être falsifiée par toute mise à jour.
    Une mise à jour particulière peut falsifier certaines contraintes alors que d'autres ne seront nullement concernées par cette mise à jour. Il faudra donc s'efforcer de trouver des critères qui permettent de déterminer les contraintes qui ne risquent pas d'être falsifiées par une mise à jour.
3.- Les contraintes d'intégrité devraient être vérifiées avant la mise à jour.
    En effet, pour tester l'effet d'une mise à jour sur les contraintes d'intégrité, on peut imaginer de réaliser la mise à

jour et d'évaluer ensuite les contraintes. Si une des contraintes est falsifiée, l'effet de la mise à jour sur la base de données devra être annulé, nous permettant de revenir à l'état initial. Cependant, cette opération "mise à jour/restauration de l'état initial" est coûteuse et il est donc préférable de l'éviter. Pour ce faire, on peut procéder de la manière suivante

- soit m la mise à jour
- soit E l'état actuel de la base de données
- soit E(m) l'état de la base de données obtenu en appliquant la mise à jour m à E
- soit C une contrainte d'intégrité.

Il s'agira de déterminer, à partir de C et de m, une contrainte S(C,m) de sorte que C sera satisfaite sur E(m) ssi S(C,m) est satisfaite sur E. La technique de vérification de contraintes d'intégrité par modification de Query [STON 76] est un exemple de mise en oeuvre de cette méthode.

4.- Nous pouvons supposer que les contraintes d'intégrité sont vérifiées sur l'état actuel de la base de données.

Par conséquent, à partir d'une contrainte C et d'une mise à jour m, nous pouvons déterminer une contrainte S(C,m) telle que S(C,m) soit vérifée sur E(m) si et seulement si C l'est. L'évaluation de S(C,m) devrait être moins coûteuse que l'évaluation de C.

5.- Les contraintes d'intégrité sont, en général, assez simples [DATE 81].

Par conséquent, nous pouvons tenter de déterminer des classes restreintes contraintes qui paraissent les plus fréquentes dans les applications et tenir compte des caractéristiques propres à ces classes pour en déduire des méthodes de mise en oeuvre efficaces. Un exemple de telles méthodes est donné dans [BERN 80].

La formalisation d'une base de données relationnelle en logique selon la vue "théorie du modèle" a permis de mettre en évidence et de valider une méthode de simplification de contraintes d'intégrité basée sur la quatrième observation, à savoir sur le fait que les contraintes d'intégrité sont supposées vraies dans l'état actuel de la base de données. Cette méthode a été définie pour le calcul relationnel à variable-domaine dans [NICO 79], [NICO 82] et pour le calcul relationnel à variable-tuple dans [BLAU 81]. Dans ce qui suit, nous présenterons la méthode dans le calcul relationnel à variable-domaine.

4.2.2 Définition de la méthode.

4.2.2.1 Présentation intuitive.

L'objectif de ce point est de permettre au lecteur de se faire une idée de la méthode avant de passer à une étude plus théorique. Comme nous l'avons déjà mentionné, la méthode consiste à tirer profit du fait que les contraintes sont satisfaites dans l'état actuel de la base de données pour en déduire une forme simplifiée. Considérons les relations et les contraintes de la figure 4.2.

Contrainte 1 : Supposons que l'on insère $\langle D,A \rangle$ dans l'extension de la relation "vend". La contrainte est vérifiée dans l'état actuel. Dès lors, si elle est falsifiée dans l'état futur, ce sera $\langle D,A \rangle$ la cause de la falsification. Il est clair qu'il suffira donc de vérifier qu'il existe une entreprise qui fournit l'article "A" au département "D" c'est à dire " $\exists z$ fournit(z,D,A)".

Contrainte 2 : De même, l'insertion de $\langle E_1, E_2 \rangle$ dans l'extension de la relation "Subord" ne nécessitera que l'évaluation de "Tous les subordonnés de E_1 sont subordonnés de E_2 et tous les supérieurs de E_2 sont supérieurs de E_3" c'est-à-dire $\forall x \forall y$ (($\neg$ subord $(x,E_1) \vee$ subord $(x,E_2)) \wedge$ (subord $(E_2,y) \vee$ subord (E_1,y)).

Les principales étapes de la méthode consistent en

(1) l'instantiation, de manière adéquate, de la contrainte de sorte que la forme instantiée soit vraie lorsque la contrainte initiale l'est.
Ainsi l'instantiation de la contrainte (1) est
$\exists z \neg$ vend(D,A) $\vee$ fournit(z,D,A) (3)

(2) l'évaluation de tout ce qui est évaluable (opérateur de comparaison) ou de ce que l'on sait "vrai" sur l'état futur. Dans la contrainte (3), nous savons que vend(D,A) sera "vrai" que " $\neg$ vrai=faux" et "faux $\vee$ A=A" ce qui ramène (3) à $\exists z$ fournit (z,D,A).

4.2.2.2 Hypothèses.

Dans ce qui suit, nous présentons les hypothèses de la méthode.

1.- Les contraintes d'intégrité sont des formules bien formées fermées sous forme normale prenexe conjonctive.

Le lecteur intéressé par la transformation d'une wff quelconque en une wff sous forme normale prenexe conjonctive peut se reporter à [CHAN 72] . Cette transformation peut ne se faire qu'une fois ; par exemple, lors de la conception du schéma de la base de données.

2. - Les contraintes d'intégrité sont des formules à champ restreint.
3. - Nous ne considérons comme mise à jour que les insertions, les suppressions et les transactions. Les modifications sont définies comme un cas particulier de transactions.
4. - Nous supposerons que les insertions et les suppressions conduisent à des suppressions effectives et des insertions effectives de tuples sans quoi les contraintes sont trivialement satisfaites, l'état de la base de données après la mise à jour étant identique à l'état avant la mise à jour.
5. - Nous supposerons que toute opération dans une transaction conduit à une insertion ou à une suppression effective de tuples. Si deux opérations ont un effet opposé, elles peuvent être supprimées puisqu'elles ne modifient pas l'état final.

4.2.2.3 Définition des contraintes à considérer.

L'objet de ce paragraphe est de donner un critère qui, étant donné un ensemble de contraintes et une mise à jour, détermine l'ensemble des contraintes qui peuvent être falsifiées par une mise à jour.

Considérons une mise à jour concernant la relation R d'arité n. Seules les contraintes contenant un symbole de prédicat n-aire R doivent être considérées. Les autres ne pourront jamais être falsifiées par cette mise à jour. Cette propriété, qui peut paraître très naturelle, n'est cependant garantie que parce que les contraintes sont des formules à champ restreint. On a montré [NICO 79] que si W est une formule à champ restreint qui est vraie dans une interprétation I, cette formule W reste encore vraie dans une interprétation I^* où I^* est l'interprétation I auquel on a ajouté une constante.

Par ailleurs, lorsqu'un tuple est inséré (resp. supprimé) dans l'extension de la relation R, seules les contraintes contenant un littéral négatif (resp. positif) dont le symbole de prédicat est R doivent être considérées. Les autres ne risquent pas d'être falsifiées par la mise à jour.

exemple : Considérons les relations

vend(départ, article)

fournit(entr, départ, article)

exprimant respectivement qu'un département vend un article et qu'une entreprise fournit un article à un département.

Considérons la contrainte

$$\forall x \, \forall y \, \exists z \quad \neg \, \text{vend}(x,y) \vee \text{fournit}(z,x,y)$$

exprimant qu'un département ne peut vendre que les articles qu'une entreprise lui fournit.

L'insertion d'un tuple dans la relation "fournit" ne falsifiera jamais la contrainte. En effet, la contrainte est vraie avant l'insertion et si l'insertion modifie le domaine sur lequel x et y prennent leurs valeurs, nous sommes néanmoins sûrs que pour ces nouvelles valeurs il existe un z telle que la contrainte est vraie, et cela quelle que soit la valeur de "$\neg$ vend(x,y)". Par contre, lorsqu'un tuple est inséré dans l'extension de la relation "vend", les domaines peuvent augmenter sans que nous soyons sûrs que la contrainte soit vérifiée pour les nouvelles valeurs.

Dans ce qui suit, nous ne considérerons que les contraintes susceptibles d'être falsifiées. Dans un premier temps, nous présenterons la méthode de simplification pour les insertions. La méthode pour les suppressions sera ensuite présentée par analogie. Nous terminerons par le résultat de la méthode pour les transactions et quelques remarques sur les formes que peuvent prendre les contraintes.

4.2.2.4 Présentation de la méthode pour les insertions.

4.2.2.4.1 Définition des substitutions.

Considérons une contrainte W.

Soit u une mise à jour qui consiste en l'insertion du tuple $\langle e_1, \ldots, e_n \rangle$ dans la relation R.

Soit I l'interprétation dans laquelle W est vraie et qui représente l'état actuel de la base de donnée.

Soit I^+ l'interprétation obtenue à partir de I en ajoutant le tuple $\langle e_1, \ldots, e_n \rangle$ à la relation R. I^+ est le nouvel état de la base de données.

Notre but consiste, bien évidemment, à nous assurer que W est encore vraie sur I^+.

L'objet de ce point (4.2.2.4.1.) est de déterminer un ensemble de substitutions $\gamma(w,u)$ tel que W est vrai dans I^+ ssi $W\gamma(w,u)$ est vrai dans I^+.

où $\mathcal{C}_{\gamma(W,U)} = W\sigma_1 \wedge \cdots \wedge W\sigma_m$ tel que
$\sigma_1, \sigma_2, \ldots, \sigma_m$ sont toutes les substitutions de $\gamma(W,u)$

A cet effet, supposons que W contienne p littéraux négatifs contenant le symbole de prédicat n-aire R. Désignons-les par

$$\text{"}\neg R(t_1^i, \ldots, t_n^i)\text{"} \quad 1 \leqslant i \leqslant p$$

Pour chacun d'eux, nous allons essayer de définir une substitution $\gamma_i (1 \leqslant i \leqslant p)$. Selon que la substitution pourra ou ne pourra pas être définie pour un littéral $\neg R(t_1^i, \ldots, t_n^i)$, cela signifera que ce littéral pourra ou ne pourra pas être à l'origine de la falsification de la contrainte.

La substitution sera définie pour un littéral $\text{"}\neg R(t_1^i, \ldots, t_n^i)\text{"} (1 \leqslant i \leqslant p)$ si et seulement si $R(t_1^i, \ldots, t_n^i)$ et $R(e_1, \ldots, e_n)$ s'unifient. Soit σ la substitution la plus générale telle que $R(t_1^i, \ldots, t_n^i)$ et $R(e_1, \ldots, e_n)$ s'unifient. La substitution γ_i est la substitution σ où l'on a supprimé tous les éléments "(c/x)" où x est une variable existentiellement quantifiée ou une variable universellement quantifiée mais qui est précédée dans la liste des quantifications par au moins une variable existentiellement quantifiée.
Autrement dit et de manière plus précise, γ_i est définie ssi

1° Si t_j^i $(1 \leqslant j \leqslant n)$ est une constante alors $t_j^i = e_j$

2° Si $t_j^i = t_k^i$ $(1 \leqslant j,k \leqslant n)$ alors $e_j = e_k$

Quand γ_i est définie, elle consiste de

$$\gamma_i = \{(e_j \mid t_j^i) \mid t_j^i \text{ est une variable universellement quantifiée non précédée par une variable existentiellement quantifiée dans l'ordre des quanticateurs de } W\varphi\}$$

Justifions ces définitions de manière intuitive.

1° Si γ_i n'est pas définie, cela signifie que quelles que soient les valeurs que pourront prendre $t_1^i, \ldots, t_n^i$, ces valeurs ne seront pas identiques à $e_1, \ldots, e_n$ c'est-à-dire que $R(t_1^i, t_2^i, \ldots, t_n^i)$ ne sera jamais égal à $R(e_1, e_2, \ldots, e_n)$.
Dès lors, $R(t_1^i, \ldots, t_n^i)$ ne sera jamais à l'origine de la falsification de W pour u.

2° Les variables existentiellement quantifiées sont exclues pour la raison suivante.
Considérons la contrainte W

$$W = q_1 x_1 q_2 x_2 \cdots q_{i-1} x_{i-1} \exists x_i q_{i+1} x_{i+1} \cdots q_n x_n F$$

où q_j $(1 \leqslant j \leqslant i-1$ et $i+1 \leqslant j \leqslant n)$ sont des quantificateurs
F est une formule bien formée sans quantificateur contenant $x_1, \ldots, x_n$ comme variables libres.

Supposons que "$(c_i|x_i)$"$\in\sigma$. Les valeurs de vérité de W et de $q_1x_1\cdots q_{i-1}x_{i-1}q_{i+1}x_{i+1}\cdots q_nx_nF\sigma'$ où $\sigma'=\{(c_i|x_i)\}$ ne seront identiques que si c_i est une valeur de x_i telle que W est vraie.

Dans le cas de variables universellement quantifiées précédées dans l'ordre des quantifications par une variable existentiellement quantifiée, rien ne nous assure non plus que la substitution donnera une formule dont la valeur de vérité est identique à celle de la formule initiale.

Considérons l'exemple suivant. La contrainte est

$$\exists y \forall x \quad (\neg P(x) \vee Q(x,y))$$

Supposons qu'elle soit vérifiée et que l'on insère P(a).

Si nous permettons que toutes les variables universellement quantifiées appartiennent aux substitutions, nous devrons vérifier la formule $\exists y \quad (\neg P(a) \vee Q(a,y))$.

Ces deux formules n'auront pas nécessairement la même valeur de vérité. Il se peut en effet qu'un y vérifie la formule simplifiée alors qu'il ne vérifie pas la contrainte initiale (par exemple s'il existe b tel que $P(b) \wedge \neg Q(b,y)$.

Remarquons enfin que $\Upsilon(W,u)$ peut être égal à l'ensemble vide $\emptyset$. Si $\Upsilon(W,u)=\emptyset$ cela signifie qu'aucun littéral ne peut être la cause de la falsification de la contrainte et donc que la contrainte est vraie dans l'interprétation I^+.

Désignons par λ la substitution vide. Il est clair que λ peut appartenir à $\Upsilon(W,u)$ lorsque $R(t_1^i,\ldots,t_n^i)$ et $R(e_1,e_2,\ldots,e_n)$ s'unifient $(1\leqslant i\leqslant p)$ et que $t_1,\ldots,t_n$ ne contient pas de variables universellement quantifiées non précédées par une variable existentiellement quantifiée.

4.2.2.4.2 Définition de $\Upsilon^+(W,u)$.

Nous avons défini $\Upsilon(W,u)$ un ensemble de substitutions telle que est vrai dans I^+ si et seulement si $W(\gamma_i)$ est vrai dans I^+ pour toute substitution γ_i de $\Upsilon(W,u)$.

Cependant, il peut se produire qu'étant donné deux substitutions γ_i et γ_j $(\gamma_i \neq \gamma_j)$, $W\gamma_j$ soit vrai dans I^+ si $W\gamma_i$ l'est.

Cela ne se produira que lorsque γ_i est plus générale que γ_j

Définition : Soit σ,σ' deux substitutions. σ est plus générale que

σ' ssi $\sigma \subset \sigma'$ c'est-à-dire si pour tout élément "$(e_i|x_i)$" il existe un élément "$(e_j|x_j)$" $\in \sigma'$ tel que $e_i = e_j$ et $x_i = x_j$.

Il est clair que, dans ce cas, seul $W\gamma_i$ devra être vérifiée.
Nous définirons $\Upsilon^+(W,u)$ comme l'ensemble $\Upsilon(W,u)$ auquel on a supprimé toutes les substitutions qui admettent dans $\Upsilon(W,u)$ une substitution plus générale $\Upsilon^+(W,u) = \{\gamma_i | \gamma_i \in \Upsilon(W,u)$ et $\nexists \gamma_j \in \Upsilon(W,u)$ $(\gamma_j \neq \gamma_i)$ tel que γ_j est plus générale que $\gamma_i\}$.

$W_{\Upsilon^+(W,u)}$ est alors défini de manière analogue à $W_{\Upsilon(W,u)}$.

exemple : Soit $W' = \forall x \forall y \neg P(x,b) \vee \neg P(x,y) \vee Q(x,y)$
Considérons l'insertion de $<a,b>$ dans P.
$\Upsilon(W,u) = \{\{(a|x)\}, \{(a|x),(b|y)\}\}$
$\{(a|x)\}$ est plus générale que $\{(a|x),(b|y)\}$
En effet si $\forall y \neg P(a,b) \vee \neg P(a,y) \vee Q(a,y)$ est vrai
alors $\neg P(a,b) \vee \neg P(a,b) \vee Q(a,b)$ est vrai.

4.2.2.4.3 Définition de la forme simplifiée $S^+(W,u)$.

Jusqu'à présent, nous avons ramené l'évaluation de W à l'évaluation $W_{\Upsilon^+(W,u)}$. Cette forme peut encore être simplifiée en tenant compte du fait que

- $R(e_1, e_2, \ldots, e_n)$ est vrai dans I^+.
- que les prédicats de type $(=, \neq, >, <, \geqslant, \leqslant)$ dont tous les arguments sont des constantes peuvent être évaluées directement et remplacées par leurs valeurs de vérité.

De même, dès que les valeurs de vérité ont été mises en évidence, les formules peuvent être simplifiées eu égard aux règles bien connues.

Comme $W_{\Upsilon^+(W,u)} = W\gamma_1 \wedge \ldots \wedge W\gamma_m$ $\gamma_1 \ldots \gamma_m$ étant les éléments de $\Upsilon^+(W,u)$, si une formule $W\gamma_i$ $(1 \leqslant i \leqslant m)$ se réduit à la valeur "faux", la contrainte sera falsifiée.

De même, si $W\gamma_i$ et $W\gamma_j$ $1 \leqslant i, j \leqslant m$ $i \neq j$ sont deux formules identiques à une permutation de disjonctions, à une permutation de littéraux et à un renommage de variable près, nous pourrons ne retenir qu'une de ces deux formules.
La formule ainsi définie constitue la forme $S^+(W,u)$.

exemple : Soit la contrainte

$$\forall x \forall y \forall z \quad \neg P(y,x) \vee \neg P(z,x) \vee (y=z)$$

Supposons que le tuple $<a,b>$ est inséré dans la relation P

$$\gamma^{+}(W,u) = \{\{(a|y),(b|x)\},\{(a|z),(b|x)\}$$

$$W_{\gamma^{+}(W,u)} = (\forall z \ \neg P(a,b) \vee \neg P(z,b) \vee (a=z))$$
$$(\forall y \ \neg P(y,b) \vee \neg P(a,b) \vee (y=a)).$$

En tenant compte du fait que P(a,b) est vrai dans I^{+}, on a

$$(\forall z \ \neg P(z,b) \vee (a=z)) \wedge (\forall y \ \neg P(y,b) \vee (y=a))$$

Nous aurons, dès lors, que l'évaluation peut se ramener à

$$\forall z \ \neg P(z,b) \vee (a=z).$$

4.2.2.4.4 Définition de la méthode en insertion : récapitulation

1° Construire l'ensemble des substitutions $\gamma(W,u)$ pour la contrainte W et la mise à jour u qui est l'insertion de $e_1,..,e_n$ dans R.
Si $\gamma(W,u) = \emptyset$ la contrainte est vraie dans I^{+}.
Si $\gamma(W,u) \ni \lambda$ alors $S^{+}(W,u) = W$

2° Construire $\gamma^{+}(W,u)$ à partir de $\gamma(W,u)$

3° Simplifier $W_{\gamma^{+}(W,u)}$ eu égard aux prédicats évaluables ou aux prédicats que l'on sait vrais sur I^{+} pour obtenir $W_{\gamma^{+}(W,u)}$.

Si $W_{\gamma^{+}(W,u)}$ peut être ramené à la valeur de vérité "faux", la contrainte est falsifiée dans I^{+} et aucun accès à la base de données n'est nécessaire.
Si $W_{\gamma^{+}(W,u)}$ peut être ramené à la valeur de vérité "vrai", la contrainte est vraie dans I^{+}et aucun accès à la base de données n'est nécessaire.
Si aucun de ces deux cas ne se présente, il faut simplifier $W_{\gamma^{+}(W,u)}$ pour éliminer les instances $W\beta_i$ de $W_{\gamma^{+}(W,u)}$ qui possèdent une instance identique aux noms de variables, à une permutation de littéraux et à une permutation de disjonctions pour obtenir $S^{+}(W,u)$.

4° $S^{+}(W,u)$ peut être évaluée par rapport au contenu de la base de données. $S^{+}(W,u)$ est vrai sur I^{+} si W est vrai sur I^{+}. Remarquons que si $S^{+}(W,u)$ ne contient que des symboles de prédicats distincts de R alors $S^{+}(W,u)$ pourra être évaluée sur I. Puisque seule l'extension de R a changé, le résultat sera identique à l'évaluation de $S^{+}(W,u)$ sur I^{+}.

4.2.2.4.5 Définition de deux formes alternatives.

La forme $S^+(W,u)$ est une forme où les variables sont instantiées au maximum. Cependant, en ce faisant, nous avons peut-être créé une conjonction d'instances $W\gamma_i$ qui peut être importante. Ce sera le cas, si la contrainte contient plusieurs littéraux qui sont impliqués dans la mise à jour. Dans ce cas, la forme $S^+(W,u)$ n'est peut-être pas la plus adéquate. Cependant, lorsque la contrainte ne contient qu'un seul littéral alors elle est sans conteste la meilleure forme que l'on peut trouver par la méthode. Selon Nicolas [NICO 79], c'est d'ailleurs ce type de contraintes qui a le plus de chances de se rencontrer. Pour les autres cas deux formes alternatives ont été définies, la préférence étant donnée à la seconde.

1. Définition de $W^+(\gamma^+,u)$

Soit γ^+ l'intersection de toutes les substitutions de $\Upsilon^+(W,u)$.
$W^+(\gamma^+,u)$ est de la forme $W\gamma^+$.

2. Définition de $T^+(W,u)$

Pour définir $T^+(W,u)$, nous définirons une suite

$$S = (S_1, S_2, \ldots, S_x) \qquad x > 0$$

à partir de $\Upsilon^+(W,u) \neq \emptyset$.

Chaque élément de la suite S_i $(1 \leq i \leq x)$ est un ensemble non vide de substitutions $\{a_1, \ldots, a_n\}$ tel que

(1) a_j $(1 \leq j \leq n) \in \Upsilon^+(W,u)$

(2) a_j $(1 \leq j \leq n) \notin S_k$ $(1 \leq k \leq i-1)$

(3) $\cap S_i$ est l'intersection de $a_1, a_2, \ldots, a_n$ et $\cap S_i \neq \emptyset$

(4) Il n'existe pas d'autres ensembles $\{b_1, b_2, \ldots, b_n\}$ répondant aux conditions (1), (2), (3) tels que $m > n$.

De plus $S_1 \cup S_2 \cup \ldots \cup S_x = \Upsilon^+(W,u)$

$$T^+(W,u) = W \cap S_1 \quad \ldots \quad W \cap S_n .$$

exemple : La contrainte

$$W = \forall x \forall y \forall z \quad \neg P(x,y,c,z,e) \vee \neg P(x,y,c,d,z) \vee \neg P(a,x,y,d,e) \vee Q(x,y,z)$$

Supposons que le tuple $\langle a,b,c,d,e \rangle$ soit inséré dans P, dès lors les substitutions possibles sont

$$\gamma_1 = \{(a|x), (b|y), (d|z)\}$$
$$\gamma_2 = \{(a|x), (b|y), (e|z)\}$$
$$\gamma_3 = \{(b|x), (c|y)\}$$

$\Upsilon^{+}(W,u) = \{\gamma_1, \gamma_2, \gamma_3\}$
$S = (\{\gamma_1, \gamma_2\}, \gamma_3)$
$\cap\{\gamma_1, \gamma_2\} = \{(a\ x), b\ y)\}$ et $\cap\{\gamma_3\} = \{(b/x), (c/y)\}$.

4.2.2.5 Application de la méthode aux suppressions.

La méthode est identique pour les suppressions si ce n'est que les substitutions sont définies en considérant les littéraux positifs au lieu des littéraux négatifs.

4.2.2.6 Application de la méthode aux transactions.

Soit u une transaction et $u_1, u_2, \ldots, u_n$ les opérations de cette transaction. Par hypothèse, l'ordre des opérations n'a aucune importance. Dès lors, pour une contrainte donnée, nous pouvons définir pour chaque opération $\Upsilon(W,u_i)$ $(1 \leqslant i \leqslant n)$. Soit $\Upsilon^{*}(W,u)$ l'union de tous les $\Upsilon(W,u_i)$ qui sont différents de l'ensemble vide. Les formes simplifiées peuvent alors être définies à partir de $\Upsilon^{*}(W,u)$ comme elles l'étaient à partir de $\Upsilon(W,u)$ dans le cas des insertions.

4.2.2.7 Sur la forme des contraintes d'intégrité.

Nous avons vu que les substitutions γ_i ne retenaient que les variables universellement quantifiées non précédées dans l'ordre des quantifications par une variable existentiellement quantifiée. Cela suggère que certaines expressions de contraintes sont plus adéquates que d'autres eu égard à la méthode de simplification. En effet, une wff admet, en général plusieurs formes normales prenexes conjonctives qui diffèrent seulement par l'ordre des quantificateurs. Par conséquent, on s'efforcera de maximiser le nombre de variables retenues dans les substitutions.
exemple : la contrainte
"$\forall x \forall y \exists z \neg$ fournit(x,y,chaussure) $\vee$ fournit(x,z,lacet)"
est plus appropriée vis-à-vis de la méthode que la contrainte
"$\forall x \exists z \forall y \neg$ fournit(x,y,chaussure) $\vee$ fournit(x,z,lacet)"

Nous remarquerons encore que lorsqu'une wff sous forme normale prenexe conjonctive contient deux conjonctions telles que les variables existentiellement quantifiées qui apparaissent dans l'une n'apparaissent pas dans l'autre et vice-versa, cette formule peut être décomposée en formules indépendantes. La formule initiale sera vraie si toutes les

formules indépendantes sont vraies. Il peut être intéressant dans certains cas de traiter des formules décomposées.

exemple : Tout article vendu par le département D_6 ou fourni par l'entreprise E_6 appartient à la classe T_3 est une contrainte qui peut s'exprimer sous forme normale prenexe conjonctive par

$$\forall x \forall y \; (\neg fournit(E_6,y,x) \vee classe(x,T_3)) \; (\neg vend(D_6,x) \vee classe(x,T_3)).$$

Elle peut être décomposée en

$$\forall x \forall y \; \neg fournit(E_6,y,x) \vee classe(,T_3)$$
$$\forall x \; \neg vend(D_6,x) \vee classe(x,T_3)$$

dont les sémantiques associées sont

- Tout article fourni par E_6 est de classe T_3
- Tout article vendu par le département D_6 est de classe T_3

Pour l'insertion d'un tuple dans la relation "fournit" par exemple, il est en effet plus intéressant de traiter avec la forme décomposée qu'avec la forme initiale.

4.2.3 Conclusion de la partie mise en oeuvre.

4.2.3.1 Evaluation de la méthode.

Par rapport au cadre de référence que nous avions énoncé, la méthode peut essentiellement être caractérisée par le fait qu'elle

- détermine quelles sont les contraintes qui risquent d'être falsifiées par une mise à jour
- tient compte du fait que les contraintes sont vérifiées dans l'état actuel de la base de données pour déterminer une forme simplifiée de ces contraintes.

Nous avons mentionné aussi que, dans le cas où la forme simplifiée ne contient pas de prédicats représentant la (ou les) relation (s) modifiée (s), la contrainte simplifiée pouvait être évaluée sur l'état actuel de la base de données. Cela signifie que pour une mise à jour donnée, les contraintes ne pourront être évaluées sur la base de données avant la mise à jour que si aucune des contraintes ne mentionne la relation modifiée dans sa forme simplifiée.

En toute généralité, nous concluerons donc que la méthode ne détermine pas des contraintes qui peuvent être appliquées avant la modification de l'état. Enfin, la méthode permet de mettre en oeuvre les optimisations développées dans le cadre des questions et peut supporter des techniques particulières comme nous le verrons ci-après. Par conséquent,

mis à part sa contribution personnelle non négligeable, la méthode a pour avantage qu'elle peut être intégrée dans un système plus complet. En ce qui concerne la méthode en elle-même, elle est très satisfaisante en ce qui concerne les insertions et les suppressions. Cependant, les transactions constituent à notre avis le point faible de la méthode (et par voie de conséquence, les modifications aussi).
En effet, la méthode n'est définie que pour des insertions et des suppressions de tuples dans une relation. Par conséquent, des opérations de la forme "supprimer toutes les entreprises qui fournissent des chaussures" nécessitent de vérifier les contraintes pour chaque entreprise qui fournit des chaussures. Peut-être est-il possible de trouver une seule contrainte simplifiée pour l'ensemble de ces suppressions.
De même, nous avions mentionné qu'une modification était exprimée sous forme d'une transaction "suppression-insertion". Dès lors, des opérations de la forme "augmenter le salaire de tous les employés de 10% peuvent être lourdes eu égard à la vérification des contraintes d'intégrité.

4.2.3.2 Extensions possibles.

L'étude de la méthode peut susciter certaines idées ou réflexions à propos d'extensions qui pourraient être intégrées à la méthode. Dans ce qui suit, nous présentons trois reflexions ; la première concerne la méthode elle-même, les deuxième et troisième sont beaucoup plus spéculatives.

La première réflexion que nous aimerions faire concerne la forme $S^+(W,u)$. Cette forme n'est en effet pas toujours minimale en ce sens qu'elle pourrait, dans certains cas, être simplifiée davantage.
En effet considérons, à titre d'exemple. la contrainte

$$\forall x \forall y \forall t_1 \forall t_2 \quad \neg P(x,b,t_1) \vee \neg P(x,y,t_2) \vee Q(x,y)$$

avec la mise à jour u qui est l'insertion de $\langle a,b,c \rangle$ dans l'extension de la relation P.
Cette contrainte admet deux instances

(1) $\forall y \forall t_2 \quad \neg P(x,y,t_2) \vee Q(a,y)$

(2) $\forall t_1 \quad \neg P(a,b,t_1) \vee Q(a,b)$

Il est clair que si la contrainte (1) est vérifiée, la contrainte (2) le sera également et, par conséquent, il n'est pas nécessaire de vérifier (2). Dès lors dans la forme $S^+(W,u) = W\gamma_1 \ \ldots \ W\gamma_p$ nous pourrions envisager de rechercher les instances $W\gamma_i$ qui sont plus générales que des instances $W\gamma_j$ $(j \neq i)$ et ne retenir que les instances les plus générales. Il n'est cependant pas évident que le coût

de cette opération soit inférieur au gain obtenu.

La deuxième réflexion concerne les domaines sur lesquels les variables prennent leurs valeurs. Si le domaine ne change pas, la contrainte étant vérifiée dans l'état précédent et la mise à jour n'apportant ou ne supprimant aucune valeur nouvelle, les variables prendront leurs valeurs sur les mêmes domaines et la contrainte sera automatiquement vraie. Mieux encore, dans la plupart des cas, seules les domaines de certaines variables sont importants et ce n'est que dans le cas où ces domaines sont modifiés que la contrainte risque d'être falsifiée.

exemple : La contrainte

"Il existe au moins un article de type T_3 fourni par toute entreprise qui fournit".

Sa forme normale prenexe conjonctive est

$$\exists x' \forall x \forall y' \forall y \forall z \quad (\text{classe}(x', T_3)) \wedge (\neg \text{fournit}(x,y,z) \vee \text{fournit}(x,y',x')).$$

Supposons l'insertion du tuple $\langle e, D, a \rangle$ dans l'extension de la relation "fournit". Remarquons d'abord que la méthode de simplification n'est d'aucune utilité le quantificateur existentiel x' excluant toute substitution différente de A.

Si l'entreprise "**e**" est déjà mentionnée dans la relation "fournit",la contrainte sera automatiquement satisfaite, étant donné qu'elle était vraie dans l'état actuel de la base de données. Il semble donc intéressant d'analyser le domaine de certains arguments avant de considérer l'évaluation de la contrainte.

Le lecteur pourra trouver dans [BUNE 79] un travail qui peut être relié à cette réflexion.

La troisième réflexion concerne l'utilisation de données redondantes pour faciliter la vérification des contraintes.

Reprenons l'exemple précédent. Supposons que l'entreprise "e" soit nouvelle. Dans ce cas, de deux choses l'une

- Soit a est un article de classe T_3 auquel cas la contrainte demande une analyse plus approfondie.
- Soit a n'est pas un article de classe T_3 auquel cas nous sommes certains qu'elle sera falsifiée.

Considérons le premier cas : si **"a"** est un article de classe "T_3 " cela entraine qu'il est encore nécessaire que "a" soit un des articles qui est fourni par toutes les entreprises. Dès lors, au lieu de réévaluer

toutes les contraintes, on peut penser à mémoriser tous les articles de classe fournis par toutes les entreprises et lors de l'insertion d'une nouvelle entreprise, vérifier si l'article est bien parmi ceux mémorisés. Il ne faut cependant pas perdre de vue que cette technique nécessite une mise à jour des informations redondantes et que le coût de mise à jour ne doit pas excéder le gain obtenu lors des vérifications de contraintes.
Une pareille technique a été développée dans [BERN 80] pour une classe restreinte de contraintes exprimées dans le calcul relationnel à variable-tuple. Pour cette classe, la méthode s'est avérée efficace. Il serait intéressant de voir jusqu'où ces résultats peuvent être étendus

4.2.4 Conclusion.

Nous avons présenté la contribution de la formalisation des bases de données selon la vue "théorie du modèle" pour l'expression et la mise en oeuvre des contraintes d'intégrité. Pour ce qui est de l'expression, nous avons indiqué que les contraintes d'état pouvaient être formulées de manière adéquate dans les langages d'interrogation de sorte que les développements élaborés au chapitre précédent restent valables.
Les contraintes de transition présentaient cependant quelques problèmes et nous avons indiqué comment on pouvait les insérer dans le cadre des contraintes d'état.
En ce qui concerne la mise en oeuvre, nous avons présenté une méthode de simplification des contraintes. L'intérêt de cette méthode, en dehors de sa contribution personnelle, résidait dans sa généralité en ce sens qu'elle pouvait incorporer des techniques développées par ailleurs. Son apport personnel consistait à exploiter le fait que les contraintes étaient satisfaites dans l'état actuel de la base de données. Nous avons indiqué que si la méthode était très intéressante pour les insertions et les tuples, les transactions constituaient son point faible et devaient être une voie de recherche ultérieure.
La formation selon la vue "théorie de la preuve" a également été utilisée dans le cadre des contraintes d'intégrité dans [HENS 84]. Leur objectif est à partir d'une mise à jour et d'un ensemble de contraintes, de définir, lors de la conception du schéma, des tests qui pourront être appliqués lors de la mise à jour sur l'état de la base de données actuel pour déterminer si les contraintes sont validées ou falsifiées.

PARTIE III : CONTRIBUTION DES AUTRES LOGIQUES.

CHAPITRE 1. LA LOGIQUE MODALE.

1.0 Introduction.

Nous avons vu qu'une base de données relationnelle faisait trois hypothèses implicites. On peut remettre en question ces hypothèses, avoir dans certains cas, envie de représenter explicitement des informations négatives, de pouvoir exprimer qu'il existe d'autres individus que ceux qui sont connus dans la base de données, de pouvoir représenter des exceptions à des règles générales et ainsi de suite.
Nous nous plaçons alors dans le cadre de bases de données incomplètes et on peut étudier d'autres logiques que la logique du premier ordre qui parait assez limitée dans ce cadre. De même, la logique ne connait que deux valeurs de vérité et dans certains cas, il peut être intéressant de ne pas donner de réponses à une question simplement parce qu'elle n'a pas de sens. Ces réflexions donc nous amène à étudier la logique modale, la logique à trois valeurs de vérité et les logiques non monotones afin de voir quel pourrait être leur impact sur les bases de données.

1.1 Définition de la logique modale.

Les logiques modales sont des extensions de la logique classique en ce sens qu'elles partagent le même langage que la logique des prédicats du premier ordre et qu'elles admettent le même ensemble de théorèmes lorsque nous nous limitons à un ensemble d'axiomes de logique du premier ordre mais qu'elles étendent le langage et permettent ainsi de déduire de nouveaux théorèmes.

La logique modale augmente la logique des prédicats pour prendre en compte les notions de possibilité et de nécessité. Très schématiquement, la logique modale est définie à partir des notions de monde et d'une relation d'accessibilité entre ces mondes.
Une affirmation est nécessaire dans un monde si elle est vraie dans tous les mondes accessibles à ce monde.
Une affirmation est possible si elle est vraie dans un monde accessible.
La sémantique de la notion de monde et la relation d'accessibilité ne sont pas définies et il sera nécessaire de les définir, lorsque l'on raisonne dans un système modal, sous peine de faire des raisonnements purement abstraits.

Pour permettre à l'exposé technique qui va suivre, d'être abordable, le lecteur peut associer à chaque monde, un individu. Le monde représente

l'ensemble des croyances d'un individu sur un domaine particulier. Chaque individu a accès aux croyances d'un certain nombre d'autres individus. La relation d'accessibilité est nécessairement réflexive, chaque individu connaissant ces propres croyances. Une affirmation est nécessaire pour un individu donné si tous les individus qu'il connait, ont cette affirmation parmi leurs croyances. Une affirmation est possible pour un individu donné s'il existe un individu qu'il connait qui possède cette affirmation parmi ces croyances.

1.1.1. Langage des logiques modales.

Le langage des logiques modales est constitué du langage de la logique du premier ordre augmenté de deux opérateurs L et M, qui peuvent être lu respectivement comme "Il est nécessaire que" et "Il est possible que", ainsi que de la règle de formation

Si P est une formule bien formée alors

LP et MP sont des formules bien formées.

Les opérateurs L et M sont les opérateurs les plus prioritaires. Ainsi dans la suite $LP \rightarrow P$ doit être interprété comme $(LP) \rightarrow P$.

1.1.2. Sémantique des logiques modales.

Nous définirons une interprétation I d'un langage modal comme un quadruplet (W,D,R,F) tel que

1) W est un ensemble non vide représentant les différents mondes possibles
2) D un domaine non vide d'éléments ou individus
3) R une relation binaire réflexive
4) F une assignation qui
   - à chaque paire (c,w) où c est une constante et w un élément de W associe un élément de D
   - à chaque paire (f^n,w) où f^n est une fonction n-aire et w un élément de W associe une fonction de $D^n \rightarrow D$ $(n > o)$
   - à chaque paire (p^n,w) où p^n est un prédicat n-aire et w un élément de W associe une fonction de $D^n \rightarrow B$ $(n \geqslant o)$ où $B = \{$ vrai, faux $\}$.

Cette définition de l'interprétation d'un langage modal permet effectivement que certains prédicats soient vrais dans certains mondes et pas dans d'autres.

La relation R, appelée relation d'accessibilité, indique quelles sont les mondes qui sont accessibles ou possibles à partir d'un monde donné. Si $(w_1, w_2) \in R$ on dira que w_2 est accessible à partir de w_1.

Nous allons maintenant définir la valeur de vérité d'une formule bien formée. Celle-ci n'est définie que par rapport à une fonction g qui assigne à chaque variable un élément de D.
La valeur de vérité d'une formule bien formée f dans un monde w par rapport à une interprétation I et une fonction d'assignation g, $\mathcal{S}_{I,g}(f,w)$ est définie par

(1) $\mathcal{S}_{I,g}(P(t_1, t_2, \ldots t_n), w) =$
$F(P,w)\ (val(t_1,w,g), \ldots, val(t_n,w,g))$

où

$val(t_i, w, g) =$ - $g(t_i)$ si t_i est une variable
- $F(f,w)\ (val(t_1',w,g), \ldots, val(t_m'w,g))$ g))
si $t_i = f(t_1', \ldots, t_m')$
- $F(t_i,w)$ si t_i est une constante.

(2) $\mathcal{S}_{I,g}(A \wedge B, w) =$ vrai si $\mathcal{S}_{I,g}(A,w) =$ vrai et $\mathcal{S}_{I,g}(B,w) =$ vrai

(3) $\mathcal{S}_{I,g}(\neg A, w) =$ vrai si $\mathcal{S}_{I,g}(A,w) =$ faux

(4) $\mathcal{S}_{I,g}(\forall x\ A, w) =$ vrai si $\mathcal{S}_{I,g\ (d/x)}^{(A,w)} =$ vrai pour tout $d \in D$

où $g\ (d/x)$ est une assignation identique à g excepté pour x qui prend la valeur d

(5) $\mathcal{S}_{I,g}(LA, w) =$ vrai si $\mathcal{S}_{I,g}(A, w') =$ vrai pour tout monde $w' \in W$ tel que $(w, w') \in R$.

Nous définissons bien sûr $MA \stackrel{def}{=} \neg(L \neg A)$.

Nous dirons qu'une wff est valide pour une classe d'interprétations définie par les propriétés de la relation d'accessibilité (réflexivité, symétrie et transitivité)si quelles que soient les mondes possibles et la fonction d'assignation, la wff est vraie dans toutes les interprétations de la classe.

Selon les propriétés que l'on exigera de la relation d'accessibilité R, les formules bien formées valides seront différentes. Ainsi si P est une formule bien formée, LP $\rightarrow$ LLP ne sera pas une formule valide si l'on exige que la relation R soit simplement réflexive mais le sera si nous exigeons que R soit transitive.

1.1.3 Théorie de la preuve.

Nous sommes en mesure de construire un système formel modal. En réalité, il existe une multitude de systèmes modaux. Les plus connus étant les systèmes T, S4, S5 définis pour la logique des propositions d'après les propriétés de la relation R.
Ces systèmes ont leurs équivalents pour la logique des prédicats et nous les appellerons, par abus de langage, également T, S4, S5.

Ces systèmes admettent tous les règles d'inférence du modus ponens et de la généralisation et de plus définissent une nouvelle règle d'inférence : "Si P est un théorème alors LP est un théorème". Cette règle est connue sous le nom de "règle de nécessité".

De plus, selon les propriétés de la relation R, nous définirons des schémas et axiomes différents. Tous les systèmes (T, S4, S5) admettent bien entendu les axiomes de la logique des prédicats du premier ordre. Pour que la théorie de la preuve corresponde à la sémantique définie, il sera nécessaire d'ajouter des axiomes qui dépendront des propriétés de la relation R. Si la relation est uniquement réflexive, il sera alors nécessaire de rajouter les axiomes

(1) $La \rightarrow a$

(2) $(L(a \rightarrow b)) \rightarrow (La \rightarrow Lb)$

(3) $(\forall x\ La) \quad (L\ \forall x a)$ (formule de Barcan)

Nous obtenons ainsi le système T. Si la relation d'accessibilité est transitive, il sera nécessaire d'ajouter l'axiome

(4) $La \rightarrow LLa$

Remarquons que cette formule peut être vraie dans certaines interprétations où la relation est uniquement réflexive. Cependant, il est possible de trouver une interprétation dans laquelle elle n'est pas vraie. Nous définissons ainsi le système S4.
De même, si la relation est symétrique, transitive et réflexive, il sera nécessaire d'ajouter l'axiome

(5) $Ma \rightarrow LMa$

définissant le système S5 McDermott [McDe 82] a montré que les systèmes T S4, S5 étaient tous complets et cohérents.

1.2 Applications de la logique modale.

C'est dans le cadre des langages de programmation et des démonstrations de programmes que la logique modale a surtout été utilisée. En effet, une instruction peut être considérée comme une fonction qui transforme un état de la mémoire d'une machine en un autre état de la mémoire de cette même machine. Les états de la mémoire représentent donc les différents mondes de la logique modale. Par conséquent, on peut définir une sémantique dénotationnelle des langages de programmation sur base des transformations d'état. De même, on peut associer à chaque instruction ou à chaque programme G les opérateurs modaux L_G et M_G et $M_G A$ (où A est une wff) sera vrai dans un état de la mémoire S s' il existe un état de la mémoire S', que l'on peut atteindre à partir de S par l'intermédiaire de G, état S' dans lequel A est vrai. La relation d'accessibilité peut être considérée comme définissant, à partir d'un état mémoire, les états mémoires accessibles grâce aux instructions du langage.

La correction partielle définie par Hoare [HOAR 69] peut être formulée en logique modale. Considérons la relation $A \{ G \} B$ exprimant que si A est vrai avant l'exécution de $\{ G \}$ alors , si l'exécution de G se termine, B est vrai dans l'état teminal. La correction partielle est démontrée si la wff "$A \rightarrow L_G B$" se voit attribuer la valeur de vérité "vrai".

De la même manière, la logique modale peut être appliquée aux bases de données soit en vue de montrer la validité d'une méthode de simplification de contraintes d'intégrité comme cela est réalisé dans [BERN 80] soit en vue de montrer qu'une transaction préserve une (ou plusieurs) contraintes d'intégrité [CASA 80]. Nous avons déjà mentionné que prouver qu'une transaction vérifie une contrainte est équivalent à montrer qu'un programme préserve un invariant.

La logique modale a également été utilisée dans les travaux de Levesque [LEVE 81] pour une formalisation des bases de données incomplètes. Dans le cadre des bases de données, on remet en cause les hypothèses du monde fermé et de fermeture du domaine. L'opérateur modal $K\alpha$ est introduit exprimant que la base de données, dans son état actuel, croit α . La base de données est vue comme ensemble d'informations partielles et est donc caractérisée par un ensemble de mondes. La relation d'accessibilté entre deux mondes exprime que ce que l'on connait ou croit dans un monde est cohérent avec ce qu'on connait ou croit dans l'

autre monde. La base de données reste cependant, de premier ordre mais le langage d'interrogation et de définition des données permet d'exprimer des formules contenant l'opérateur K . Ces formules peuvent être utilisées pour exprimer que tous les tuples d'une relation sont connus. Par exemple, pour la relation professeur (x) on peut exprimer

$$\forall x \text{ professeur } (x) \rightarrow K \text{ professeur } (x) \qquad (1)$$

l'opérateur K est résolu par rapport au contenu de la base de données et si a,b,c sont les seuls tuples de la relation 'professeur', la base de données contiendra

$$\forall x \text{ professeur } (x) \rightarrow x = a \vee x = b \vee x = c$$

ce qui est un axiome de complétude.

De même, l'information "Il existe un professeur que je ne connais pas"

$$\exists x \text{ professeur } (x) \wedge \neg K \text{ professeur } (x) \qquad (2)$$

aurait donné lieu à l'insertion de l'axiome

$$\exists x \text{ professeur } (x) \wedge \neg (x = a \vee x = b \vee x = c)$$

L'insertion de (2) dans la base de données n'aurait plus jamais permis de dire que maintenant tous les professeurs sont connus. En effet, l'insertion de (1) aurait produit une incohérence.

Remarquons cependant que les résultats obtenus sont fortement théoriques et de nombreux concepts pratiques devraient être élaborés avant de conduire à une utilisation dans le contexte des bases de données.

CHAPITRE 2. LA LOGIQUE A TROIS VALEURS DE VERITE.

2.1 Définition des différentes logiques.

Les logiques classiques utilisent deux valeurs de vérité dans la définition de leur sémantique : la valeur "vrai", notée "t", et la valeur "faux", notée "f". Cependant, cette restriction peut être contraignante et certains auteurs, pour des raisons parfois fort diverses, ont proposé des logiques à trois valeurs de vérité : "t","f",'i". Correspondant à la sémantique qu'ils accordaient à la valeur "i", ces auteurs ont défini des logiques différentes et dans ce qui suit, nous présenterons les logiques définies par Lukasiewicz, Bochvar et Kleene. Ces logiques sont des logiques déviantes en ce sens qu'elles utilisent le même langage que les logiques classiques mais diffèrent quant aux théorèmes qu'elle admettent. Ainsi le principe du tiers-exclus "P $\vee$ $\neg$ P" n'est plus un théorème de ces logiques.

La logique de Lukasiewicz a été introduite pour le traitement des affirmations contingentes concernant le futur. La signification de la troisième valeur de vérité est "indéterminé". Les tables de vérité de la logique de Lukasiewicz sont données par :

A	$\neg$A
vrai	faux
i	i
faux	vrai

A $\wedge$ B

A B	vrai	i	faux
vrai	vrai	i	faux
i	i	i	faux
faux	faux	faux	faux

A $\vee$ B

A \ B	vrai	i	faux
vrai	vrai	vrai	vrai
i	vrai	i	i
faux	vrai	i	faux

A $\rightarrow$ B

A \ B	vrai	i	faux
vrai	vrai	i	faux
i	vrai	vrai	i
faux	vrai	vrai	vrai

On remarquera que A$\rightarrow$B n'est plus équivalent à $\neg$A$\vee$B. D'autre part, la valeur de vérité A$\rightarrow$B est vrai si A et B ont la valeur "i". Cela permet de conserver le principe d'identité P$\rightarrow$ P. La justification de Lukasiewicz pour l'introduction de la troisième valeur de vérité est la suivante. Si nous ne permettons pas aux affirmations concernant le

futur de prendre une valeur de vérité distincte de "t" et "f", nous tombons dans le fatalisme. Considérons l'affirmation "Je serai à Varsovie le 21 décembre à minuit". Si cette affirmation est vraie aujourd'hui, alors je ne peux pas ne pas être à Varsovie le 21 décembre. De manière identique, si cette affirmation est fausse aujourd'hui, je ne peux pas être à Varsovie le 21 décembre. Dès lors, selon que l'affirmation soit vraie ou fausse aujourd'hui, il est nécessaire ou impossible que je sois à Varsovie le 21 décembre. Par conséquent, pour Lukasiewicz, la seule manière d'échapper au fatalisme est de ne pas imposer de valeur de vérité ("t" ou"f") à l'affirmation. C'est pourquoi Lukasiewicz introduit la valeur "i".
Cependant, l'argumentation de Lukasiewicz est critiquée par Susan Haack [HAAC 78] car elle contredit les principes de la logique modale. Le raisonnement de **Lukasiewiez** revient à passer de $L(a \rightarrow b)$ à $(a \rightarrow Lb)$ et $L(a \rightarrow b) \longrightarrow (a \rightarrow Lb)$ n'est pas une formule valide.

La logique de Bochvar donne à la valeur de vérité "i" la signification "paradoxale" ou "dépourvu de sens". Elle a été introduite pour résoudre le problème des paradoxes. Par exemple,"Cette affirmation est fausse" est vraie si elle est fausse et est fausse si elle est vraie. La proposition de Bochvar concernant ce paradoxe était de ne pas attribuer la valeur "vrai" ou la valeur "faux" mais bien la valeur "paradoxal". Correspondant à cette sémantique, Bochvar a construit les tables de vérité suivantes :

A	⌐A
vrai	faux
i	i
faux	vrai

$A \wedge B$

A \ B	vrai	i	faux
vrai	vrai	i	faux
i	i	i	i
faux	faux	i	faux

$A \vee B$

A \ B	vrai	i	faux
vrai	vrai	i	vrai
i	i	i	i
faux	vrai	i	faux

$A \rightarrow B$

A \ B	vrai	i	faux
vrai	vrai	i	faux
i	i	i	i
faux	vrai	i	vrai

A	TA
vrai	vrai
faux	faux
i	faux

(opérateur d' assertion)

connecteurs externes

$\neg' A = \neg TA$ $\qquad A \rightarrow' B = TA \rightarrow TB$

$A \wedge' B = TA \wedge TB$

$A \vee' B = TA \vee TB$

Aux tables classiques, il a ajouté un opérateur T, appelé opérateur d'assertion qui permet de revenir à une logique à deux valeurs de vérité et dont la sémantique est "Il est vrai que". On peut y voir que chaque fois qu'une sous-formule s'évalue à "i", la formule qui la contient s'évalue aussi à "i".
La logique de Bochvar n'a pas résolu le problème des paradoxes. En effet, l'affirmation "Cette affirmation est fausse ou paradoxale "est vraie quand elle est fausse ou paradoxale et est fausse ou paradoxale quand elle est vraie. Cependant elle a trouvé d'autres applications qui sont intéressantes. Une de ces applications est le traitement des présuppositions. Une wff est en effet, dans certains cas composée d'une partie "présupposition" et d'une partie qui donnera la valeur de vérité de la formule lorsque la présupposition est vraie. Cependant, lorsque la présupposition est fausse, la wff n'est ni vraie ni fausse mais plutôt "dépourvue de sens". C'est pourquoi il semble que la logique de Bochvar soit adéquate pour ce type d'applications.

Enfin, Kleene a introduit une logique pour traiter des propositions ou des formules indécidables exprimant par là, un état de partielle ignorance. Les formules indécidables recevaient par conséquent, la valeur "i" qui signifiait "indécidable".
La logique de Kleene est celle qui diffère le moins de la logique classique. Ses tables de vérité sont semblables à celles de Lukasiewicz pour les connecteurs "$\neg$","$\wedge$","$\vee$". La table de vérité pour l'implication est donnée par

$A \longrightarrow B$

A \ B	vrai	i	faux
vrai	vrai	i	faux
i	vrai	i	faux
faux	vrai	vrai	vrai

Une disjonction de littéraux $L_1 \vee L_2 \vee \ldots \vee L_n$ s'évalue à "vrai" dès que l'on peut trouver un L_i $(1 \leqslant i \leqslant n)$ qui s'évalue à "vrai". De même, une conjonction de littéraux $L_1 \wedge L_2 \wedge \ldots \wedge L_m$ s'évalue à "faux" dès que l'on peut trouver un L_i $(1 \leqslant i \leqslant m)$ qui s'évalue à "faux". Malgré cela, l'approche de Kleene peut être critiquée. En effet, si la valeur "i" est assignée à la proposition P, la formule $P \vee \neg P$ s'évaluera à "i" alors

que dans ce cas, nous aimerions la voir s'évaluer à "vrai". Cependant, pour obtenir une telle évaluation, il est nécessaire que la valeur d'une formule ne soit pas fonction de la valeur de ses sous-formules. Correspondant à chacune des logiques définies précédemment, on peut redéfinir la notion d'interprétation.

2.2 Application des logiques à trois valeurs de vérité.

Cood [COOD 79] a utilisé une logique à trois valeurs de vérité pour le traitement des valeurs nulles dans les bases de données relationnelles. Les tables de vérité de sa logique sont identiques à celles de Kleene et la valeur "i" signifie "inconnu". De plus, les opérations de l'algèbre relationnelle et les opérations de comparaison ont été généralisées pour prendre en compte les valeurs nulles. Ainsi, une opération de comparaison contenant une valeur nulle reçoit la valeur de vérité "i". Une des critiques de l'approche de Cood pour le traitement des valeurs nulles est identique à la critique que nous avons mentionnée pour la logique de Kleene.

L'interrogation d'une base de données réclame dans certains cas, une logique à trois valeurs. Cette constation s'est surtout révélée dans l'interrogation d'une base de données en langage naturel [COLM 79] [COLM 81] [DALH 79] [DALH 82].En effet, il est fréquent dans ce cadre, de faire des suppositions. Ainsi, la question "Le chat que Sophie tient dans ses bras miaule-t-il ?" recevra la valeur de vérité "vrai" ou "faux" lorsque Sophie tient un chat dans ses bras selon que celui-ci miaule ou non. Cependant, quelle valeur lui accorder lorsque Sophie ne tient pas de chat dans ses bras ? Bien qu'en logique standard ou en mathématique la valeur de vérité est "vrai" dans le cadre des bases de données, la valeur de vérité ne devrait être ni "vrai" ni "faux" mais plutôt "absurde" ou "dépourvu de sens" exprimant par là qu'une supposition n'est pas vérifiée.

De même, une question de la forme "Est-ce que tous les fournisseurs de chaussures fournissent des lacets ?" contient une présupposition implicite à savoir "Il existe au moins un fournisseur de chaussures". Dans le cas où il n'existe pas de fournisseurs de chaussures, la réponse à la question, dans le cadre des bases de données, ne devrait être ni vraie ni fausse mais plutôt notre troisième, valeur. Une constatation identique a d'ailleurs donné lieu aux opérations de vérification en FQL [PIRO 77].

C'est ainsi que Colmerauer et Pique [COLM 79] ont défini une logique à trois valeurs pour le traitement du langage naturel pour l'interrogation d'une base de données. Cette logique est une mise en oeuvre de la logique de Bochvar. L'intérêt de cette approche est que puisque le domaine d'interprétation considéré est fini, l'opérateur d'assertion peut être utilisé pour ramener les déductions dans la logique ainsi définie dans une logique à deux valeurs. Ainsi, pour une question fermée, on s'efforcera dans un premier temps de montrer qu'elle est vraie. Si elle ne l'est pas, on tentera de montrer qu'elle est fausse. Si elle n'est ni vraie ni fausse, c'est qu'elle contient une présupposition qui n'est pas vérifiée.

CHAPITRE 3. LES LOGIQUES NON-MONOTONES.

3.1 Introduction.

Une des critiques les plus pertinentes concernant la logique des prédicats du premier ordre en tant que formalisme de représentation de connaissances est son caractère monotone. Les règles d'inférence sont telles que les axiomes sont exclusivement permissifs en ce sens que l'ajout d'un nouvel axiome permet uniquement de déduire de nouveaux théorèmes sans jamais en supprimer. Il n'existe en effet, aucun moyen d'ajouter de l'information pour dire au système logique quelles inférences il ne devrait pas effectuer [MINS 75].

Une logique non monotone est une logique où l'introduction de nouveaux axiomes peut invalider les théorèmes déduits précédemment et qui permet, par conséquent, aux axiomes d'être également restrictifs. Les logiques non monotones sont particulièrement adaptées pour la modélisation des raisonnements de bon sens, pour la révision des connaissances ainsi que les informations incomplètes [McDE 80].

L'idée générale à la base des recherches en logique non monotone est l'introduction d'une règle d'inférence de la forme

"P est un théorème si $q_1, q_2, \ldots, q_n$ ne sont pas des théorèmes."

Cela permet l'expression de connaissances de la forme

"La plupart des oiseaux volent."

qui peut également s'exprimer sous la forme

"Si x est un oiseau et je ne peux pas prouver que x ne vole pas, alors x vole."

De cette information et de l'information "a est un oiseau", il est possible de déduire que "a vole". Cependant, si nous savons que "les pingouins ne volent pas" l'ajout de l'information "a est un pingouin" invalidera le théorème précédent "a vole" manifestant par là un comportement non monotone.

On définira alors un nouvel opérateur M, que l'on peut lire comme "Il est **cohérent** que " et une nouvelle règle d'inférence qui peut s'énoncer intuitivement par

$$\text{Si } A \nvdash \neg P \text{ alors } A \vdash MP.$$

La propriété de monotonie ou de non-monotonie pour une logique est une propriété plutôt syntaxique. C'est la raison pour laquelle il n'est pas surprenant qu'il n'existe pas une logique non monotone et qu'il existe des logiques non monotones qui aient des bases logiques assez différentes. Dans ce qui suit, nous distinguerons essentiellement les logiques

par défaut et les logiques autoépistémiques et nous nous concentrerons sur les logiques autoépistémiques. Nous présenterons les travaux de Moore [MOOR 84] et ceux de Doyle et McDermott. [McDE 80] [McDE 82] Nous terminerons par l'apport des logiques non monotones aux bases de données.

3.2 Logique par défaut versus logique autoépistémique.

3.2.1 La logique par défaut.

La logique par défaut consiste à effectuer des inférences à partir de suppositions qui sont justifiées par l'absence d'informations contraires. Elles permettent ainsi de définir des extensions à une théorie du premier ordre, extensions que l'on espère conformes à la réalité. Cependant, la sémantique de l'opérateur M n'est pas définie à l'intérieur de la logique et les résultats obtenus en appliquant les règles d'inférence aux axiomes d'une théorie ne sont pas des théorèmes de logique du premier ordre. Les inférences que l'on effectue ne sont en aucun cas valides en ce sens qu'il ne suffit pas que les axiomes soient vrais pour que les théorèmes que l'on pourra en déduire le soient également. Considérons les informations suivantes

(1) "a est un oiseau"

(2) "la plupart des oiseaux volent"

Nous pourrons conclure de ces axiomes que "a vole". Cependant, les axiomes (1) et (2) peuvent être vrais sans que pour autant l'affirmation "a vole" le soit.

3.2.2 La logique autoépistémique.

La sémantique de l'opérateur M n'est pas définie à l'intérieur d'une logique par défaut. Il permet simplement de définir des extensions à une théorie, par exemple, du premier ordre. Dans une logique autoépistémique, la sémantique de l'opérateur M et de son dual ¬ L ¬ est définie à l'intérieur même de la logique et il est donc possible de donner une sémantique précise à ces opérateurs. On peut ainsi définir une logique particulière, adaptée au traitement de la non-monotonie, pour laquelle on se donne une sémantique et on lui associe une théorie de la preuve lui correspondant. Cette approche nous paraît préférable et plus ambitieuse. La logique autoépistémique est une logique de croyances et l'opérateur L peut être lu comme "Je crois que".

Dans cette logique, l'information "la plupart des oiseaux volent" exprimée par

$$\forall x \text{ oiseau}(x) \wedge M\text{vole}(x) \longrightarrow \text{vole}(x)$$

signifiera

"Si x est un oiseau et s'il est **cohérent** avec l'ensemble de nos croyances d'affirmer que x vole, alors x vole."

L'opérateur M correspondra donc à la sémantique suivante : MP est vrai ssi ¬P n'est pas dérivable de l'ensemble de nos croyances. La supposition implicite réalisée dans la logique autoépistémique est que nous connaissons tous les oiseaux qui ne volent pas et que, dès lors, les seuls oiseaux qui ne volent pas sont ceux pour lesquels on peut prouver qu'ils ne volent pas. Si "a est oiseau" et qu'il ne nous est pas possible de prouver que "a ne vole pas", comme nous connaissons tous les oiseaux qui ne volent pas, nous pouvons conclure que "a vole". Dans cette logique de croyances, l'inférence "a vole" sera une inférence valide puisqu'étant donné mon ensemble de croyances, il est normal que "a vole" soit vrai. La forme de raisonnement développée ici est donc un raisonnement sur nos propres connaissances ou nos propres croyances.

3.2.3 La non-monotonie dans les deux approches.

Ces deux logiques sont des logiques non monotones mais pour des raisons différentes.
La logique par défaut est non-monotone car ses inférences ne sont en aucun cas définitives et l'ajout de nouvelles informations (de nouveaux axiomes) peut remettre en cause les inférences réalisées précédemment. La logique autoépistémique est non-monotone en raison de sa sémantique indexée. Les inférences dans cette logique sont valides et ne seront pas remises en question mais la sémantique de l'opérateur M dépend de l'ensemble des croyances. Si ces croyances augmentent (on ajoute de nouveaux axiomes), la sémantique de l'opérateur M se modifie et les théorèmes vrais par rapport à l'ensemble précédent de croyances peuvent ne plus l'être dans le nouvel ensemble de croyances.

3.3 La logique autoépistémique de Moore.

Une logique **autoépistémique** est une logique de croyances. On s'intéresse à l'ensemble de croyances (théorèmes) qu'un agent idéalement rationnel pourrait déduire d'un ensemble initial de croyances (Premisses). L'agent est idéalement rationnel en ce sens qu'il est conscient de la

totalité de ces croyances. La logique autoépistémique considère principalement l'opérateur L, M P étant défini comme ¬ L ¬ P. La proposition L P doit être lue "On croit que P". Moore se restreint à la logique des propositions dont il augmente le langage de l'opérateur L.

L'ensemble de toutes les croyances d'un individu est appelé théorie autoépistémique.

3.3.1 Sémantique d'une théorie autoépistémique.

Une interprétation propositionnelle d'une théorie T est une assignation de valeur de vérité identique à celle de la logique des propositions et telle que les formules de la forme L P où P est une formule , reçoivent une valeur de vérité quelconque.
Un modèle propositionnel d'une théorie autoépistémique T est une interprétation propositionnelle de T dans laquelle toutes les formules de T sont vrais.

Une interprétation autoépistémique d'une théorie T est une interprétation propositionnelle de T dans laquelle pour toute formule P, L P est vraie si et uniquement si P est dans T.
Un modèle autoépistémique d'une théorie T est une interprétation autoépistémique dans laquelle toutes les formules sont vraies.
Cette sémantique formelle correspond à la sémantique intuitive que nous accordons à l'opérateur L. Remarquons d'abord que la valeur de vérité de Lp ne dépend aucunement de la valeur de vérité de P mais simplement de la présence ou de l'absence de P dans la théorie autoepistémique. LP peut appartenir à la théorie et P peut être faux.
Considérons une théorie autoépistémique T qui représente les croyances d'un individu dans un monde donné. Ce monde définit une assignation de valeur de vérité aux propositions de cette théorie. Si une formule est dans cette théorie et s'il existe une formule LP dans T la valeur de vérité de LP est vrai. Dès lors,si toutes les formules de T sont vraies, le monde est un modèle de T et toutes les croyances de l'agent sont vraies.
Il nous faut maintenant inclure la notion d'inférence dans notre logique. Elle correspondra à la notion sémantique : "Quelles sont les croyances qu'un agent idéalement rationnel adopterait sur base d'un ensemble initial de croyances ?".

Une théorie autoépistémique T est valide par rapport à un ensemble de premisses A ssi toute interprétation autoépistémique de T dans laquelle

les premisses sont vraies est un modèle de T.
Cette notion de validité est la plus faible condition assurant que les croyances d'un agent soient vraies si les premisses sont vraies.

Une théorie autoépistémique T est sémantiquement complète si et seulement si T contient toutes les formules qui sont vraies dans tout modèle autoépistémique de T.
Cette notion exprime le fait que l'agent est idéalement rationnel.

3.3.2 Théorie de la preuve.

La sémantique de la logique autoépistémique étant définie, il convient de déterminer des critères syntaxiques qui caractérisent les théorèmes d'une telle logique.
Cependant, dans le cadre des logiques non-monotones, la caractérisation des théorèmes n'est pas chose aisée. En effet, les logiques non-monotones ajoutent aux règles d'inférences des théories du premier ordre, une nouvelle règle définie formellement par

" Si $A \not\vdash \neg P$ alors $A \vdash_{MP}$ ".

Cependant cette règle d'inférence est définie en fonction de la notion de dérivabilité et dès lors, ne constitue pas une définition adéquate pour une règle d'inférence. Plus généralement, l'ensemble des théorèmes qui peuvent être dérivés à partir des deux règles d'inférence de modus ponens et de généralisation ainsi que de la nouvelle règle d'inférence ne pourra plus être défini par un processus itératif aussi simple que pour les théories du premier ordre. La raison en est que les "théorèmes" générés à l'étape i peuvent invalider les "théorèmes" générés à l'étape j ($j<i$). Dès lors, l'ensemble des théorèmes d'une théorie non-monotone de premier ordre sont caractérisés par des définitions "point fixe" qui ne donnent pas directement un algorithme pour les énumérer.
Pour ce qui est de la logique autoépistémique, la caractérisation va se faire à l'aide de deux conditions exprimant respectivement les notions sémantiques de complétude et de validité d'une théorie autoépistémique.
D'abord, pour être complète, une théorie autoépistémique T doit être stable, ce qui signifie vérifier les trois coditions suivantes :

(1) Si $P_1, P_2, \ldots, P_n \in T$ et $P_1, P_2, \ldots, P_n \vdash Q$ alors $Q \in T$
où "$\vdash$" exprime la déduction à l'aide de modus ponens.
(2) $LP \in T$ si $P \in T$.
(3) $\neg LP \in T$ si $P \notin T$.

Moore [MOOR 84] a montré qu'une théorie autoépistémique était stable si et seulement si elle est sémantiquement complète.

Cependant, ces conditions ne nous apprennent pas quelles sont les croyances qu'un agent doit ou ne doit pas adopter.
Considérons les premisses $\neg LC \rightarrow D$ et $\neg LD \rightarrow C$ qui sont les correspondants de $MC \rightarrow \neg D$ et $MD \rightarrow \neg C$ La théorie qui contiendrait **LC, LD, C, D** est stable mais ne correspond pas à la sémantique définie précédemment. Dès lors, pour être valide par rapport à un ensemble de prémisses A, une théorie autoépistemique doit être basée sur A.

Une théorie autoépistémique T est basée sur un ensemble de premisses A si et seulement si toute formule de T est une conséquence logique (à l'aide de modus ponens) de $A \cup \{LP \mid P \in T\} \cup \{\neg LP \mid P \notin T\}$.
Moore [MOOR 84] a montré qu'une théorie autoépistémique était valide par rapport à un ensemble de premisses A si elle était basée sur A.

Ainsi, la théorie T qui contiendrait **LD, LC, C, D** ne serait pas basée sur les prémisses $\neg LC \rightarrow D$ et $\neg LD \rightarrow C$ car pour justifier la présence de C et D, il faudrait que $\neg LC$ et $\neg LD$ soient présents. Seules les théories qui contiendraient soit **LD** et $\neg C$ soit LC et $\neg D$ seraient basées sur ces premisses.

De même aucune théorie ne sera basée sur $\neg LP \rightarrow P$. En effet, si elle contient P, pour être basée sur $\neg LP \rightarrow P$ elle doit contenir $\neg LP$, ce qui est impossible. Si elle ne contient pas P, alors elle contient $\neg LP$ et donc **P**, ce qui est contradictoire.

On remarquera, par ailleurs, que la non-monotonie est introduite par des formules de la forme $\neg LA$ où A est une formule. En effet, l'introduction de nouvelles croyances initiales (axiomes) pourra supprimer des formules de cette forme exhibant par là un comportement non-monotone.

Le dernier problème à **traiter** est de déterminer quel est l'ensemble des théorèmes qui découlent d'un ensemble initial de prémisses. Lorsqu'il existe une et une seule théorie basée sur cet ensemble, la réponse est évidente. Mais que se passe-t-il s'il en existe plusieurs ou s'il n'en existe **pas**. Dans le cadre de la sémantique définie ici, cette question ne pose pas de problèmes. S'il existe plusieurs théories elles représentent des ensembles de croyances que l'agent peut adopter à partir d'un

ensemble initial de croyances. L'agent est libre de croire chacun de ces ensembles mais pas plusieurs simultanément. De plus, il ne dispose pas de suffisamment d'informations pour adopter un de ces ensembles plus qu'un autre. Lorsqu'il n'existe pas de théorie stable basée sur l'ensemble des premisses, cela signifie simplement que les premisses étant ce qu'elles sont, on ne peut rien en déduire.

3.4 La logique non-monotone de Doyle et McDermott.

Les travaux de Doyle et McDermott ont largement contribué au développement des logiques non monotones. Leur objectif n'était pas uniquement de définir une logique par défaut mais plutôt de donner une base sémantique au concept de non-monotone et de définir une logique que Moore a appelé depuis autoépistémique. Pour ce faire, ils ont tenté de modifier la logique des prédicats du premier ordre pour y inclure la notion de non monotonie [McDE 80]. Cependant, le manque de sémantique derrière cette proposition a amené McDermott à se placer dans le cadre des logiques modales et de modifier sa sémantique et sa théorie de la preuve afin de capturer la non-monotonie [McDE 82]. Dans ce qui suit, nous présenterons ces deux propositions et nous montrerons en quoi elles ne sont pas satisfaisantes.

3.4.1 Modification de la logique des prédicats du premier ordre.

Nous savons que l'ensemble des théorèmes d'une théorie du premier ordre est récursivement énumérable et qu'il existe un algorithme simple pour réaliser cette énumération. En effet, partant d'un ensemble S_o de formules qui contient les axiomes de la théorie, on peut appliquer les règles d'inférence pour obtenir un ensemble S'_o. Définissons $S_1 = S_o \cup S'_o$. On peut appliquer les règles d'inférence pour obtenir un ensemble S'_1 définissant un nouvel ensemble S_2 et ainsi de suite. Une formule est un théorème si elle appartient à un ensemble S_i ($i \geq 0$). Cette manière de procéder est rendue possible par le fait qu'aucun théorème généré à l'étape i ne remettra en cause un théorème généré à l'étape j ($i > j$).

De plus, les théories du premier ordre satisfont la propriété d'idempotence. Si A est une théorie et si Th(A) représente l'ensemble des théorèmes de cette théorie $Th(A) = \{ P : A \vdash P \}$ alors $Th(A) = Th(Th(A))$.

Cette dernière propriété peut être exprimée de la manière suivante. L'ensemble des théorèmes d'une théorie du premier ordre est le point fixe d'un opérateur qui calculerait la fermeture transitive d'un ensemble de formules à l'aide des règles d'inférence et Th(A) est le plus petit

point fixe de cet opérateur.
Dans [McDE 80] , le langage de la logique des prédicats du premier ordre est augmenté par l'opérateur unaire M. Afin de caractériser l'ensemble des théorèmes d'une théorie-non monotone de premier ordre A, la notion de dérivabilité de la logique des prédicats du premier ordre est conservée et un opérateur NM_A (NM pour non-monotone) est défini comme suit.

Soit S un ensemble de formules de L,

$NM_A(S) = Th(A \cup As_A(S))$

où $As_A(S)$ est l'ensemble des suppositions autorisées par S et est défini par $As_A(S) = \{ M\Omega : \Omega \in L \text{ et } \Omega \notin S \} \setminus Th(A)$.

$Th(A \cup As_A(S))$, (respectivement $Th(A)$), définit l'ensemble des théorèmes qui peuvent être dérivés par les règles d'inférence de la logique des prédicats du premier ordre pour la théorie $A \cup As_A(S)$ (resp A).

Par analogie avec la définition "point fixe" d'une théorie du premier ordre, nous aimerions que l'ensemble des théorèmes d'une théorie non-monotone A soit le plus petit ensemble X tel que $X = NM_A(X)$ c-à-d que $Th_{NM}(A)$ soit le plus petit point fixe de NM_A.

Exemple.

Considérons la théorie non-monotone A définie par les axiomes propres

$$\forall y \text{ oiseau}(y) \wedge M \text{ vole}(y) \rightarrow \text{vole}(y)$$
$$\text{oiseau}(\text{titi}) \leftarrow$$

Cette théorie admet un plus petit point fixe qui contiendra les formules M vole(titi) et vole(titi).

Cependant, il est très possible qu'une théorie A admette plusieurs points fixes mais que l'intersection de ces points fixes soit l'ensemble vide ou même qu'une théorie n'admette pas de points fixes.
Dès lors, il n'est pas possible de définir $Th_{NM}(A)$ comme le plus petit point fixe de NM_A.

exemple 1 : Considérons deux propositions D et C.
Considérons la théorie A dont les axiomes propres sont

$$MC \rightarrow \neg D$$
$$MD \rightarrow \neg C$$

Cette théorie admet deux points fixes F_1, F_2. Le premier contient MC et $\neg$ D et le second MD et $\neg$ C. Cependant, l'intersection de ces deux points fixes n'est pas un point fixe. En effet, $F_1 \cap F_2$ ne contient ni

$\neg C$ ni $\neg D$ de sorte que $NM_A (F_1 \cap F_2)$ contient MC et MD et donc $\neg D$ et $\neg C$ Dès lors, $F_1 \cap F_2 \subset NM_A (F_1 \cap F_2)$ et $F_1 \cap F_2$ n'est pas un point fixe. De même l'union $F_1 \cup F_2$ n'est pas un point fixe car $F_1 \cup F_2$ contient $\neg C$ et $\neg D$ et donc $NM_A (F_1 \cup F_2)$ ne contient pas MC et MD et $NM_A (F_1 \cup F_2) \subset F_1 \cup F_2$. Ce dernier résultat indique que l'ensemble des théorèmes ne peut pas non plus être l'union des points fixes.

exemple 2 : Considérons la théorie A dont l'axiome propre est

$$MC \rightarrow \neg C$$

Cette théorie n'admet pas de point fixe. En effet, supposons que P soit un point fixe qui contienne $\neg C$. Comme la seule justification pour avoir $\neg C$ est la présence de MC, F contient MC d'où contradiction. Si F ne contient pas $\neg C$ alors $NM_A (F)$ contient MC et donc $\neg C$ contradiction.

Par conséquent, nous définirons l'ensemble des théorèmes d'une théorie non monotone A de premier ordre par

$Th_{NM}(A)$ = l'insertion de tous les points fixes si cette intersection est non vide
le langage L sinon

Dans le cas où $Th_{NM} (A) = L$ alors la théorie A est inexistante.
Dans de nombreux cas, les théorèmes définis comme précédemment correspondront à la sémantique intuitive que nous accorderons à l'opérateur M et à la nouvelle règle d'inférence. Cependant, il existe un certain nombre de cas paradoxaux.
Ainsi, la théorie non monotone A dont les axiomes propres sont MP et $\neg P$ admettra un plus petit point fixe contenant MP et $\neg P$ et donc cette théorie est cohérente. Cela signifie que nous pouvons à la fois croire que P est faux et que P est cohérent avec l'ensemble de nos croyances. Manifestement, cette théorie ne devrait pas être cohérente. Cela met en évidence le principal défaut de cette logique non monotone : le manque de définition sémantique. Plus précisément, Mc-Dermott et Doyle donne une définition sémantique mais elle se réfère à la caractérisation des théorèmes. En ce sens, leur logique se rapproche plus d'une logique par défaut alors que leur objectif est de définir une logique autopistémique.

3.4.2 Modifications des logiques modales.

Conscient de la faiblesse sémantique de cette première proposition, Mc-Dermott [McDE 82] s'est alors placé dans le cadre des logiques modales pour définir à la fois une sémantique et une théorie de la preuve à sa logique non monotone. Il a, ainsi, modifié la sémantique et la théorie de la preuve des logiques modales et a analysé dans quel système (T, S4, S5) il serait plus adéquat de se placer.
Bien entendu, la sémantique de l'opérateur M des logiques modales vues précédemment n'a plus rien à voir avec la sémantique de l'opérateur M défini dans ce cadre.
Cette fois, la notion de dérivabilité fait référence à la déduction en logique modale et l'ensemble des théorèmes est caractérisé par

- Soit A une théorie modale du premier ordre
- Soit S un ensemble de formules d'un langage modal L

$$NM_A\ (S) = Th\ (A \cup As_A\ (S))$$

où

$$As_A\ (S) = \{\ P : P \in L \text{ et } P \notin S\ \} - Th\ (A).$$

$$Th = \{P : X \vdash P\}$$

où "$\vdash$" est la notion de dérivabilité modale sous T ou S4 ou S5.

Correspondant à cette caractérisation selon la théorie de preuve, Mc Dermott modifie la sémantique des logiques modales.
Il reste alors à choisir quel système est le plus approprié pour modéliser le comportement non monotone.
Examinons dans un premier temps les axiomes modaux ajoutés par S4 et S5. L'axiome $LP \rightarrow LLP$ (S4) ne fait rien d'autre qu'exprimer la deuxième condition de stabilité. De même, l'axiome $MP \rightarrow LMP$ est équivalent à $\neg LP \rightarrow L \neg LP$ qui exprime quant à lui la condition (3). Il n'existe donc aucune raison de croire que le système S5 n'est pas adéquat pour servir de base à une logique non monotone. Malheureusement, il se révèle que l'ensemble des théorèmes que l'on peut dériver sous S5 d'une théorie modale non-monotone est identique à l'ensemble des théorèmes d'une théorie modale monotone, toujours sous S5.
Cela signifie que la sémantique que l'on accorde à l'opérateur M ne correspond pas à notre sémantique intuitive.

exemple : Soit la théorie A dont l'axiome propre est

$$MC \rightarrow C$$

qui signifie "si rien ne dit que C est faux alors C est vrai". Cette théorie admet deux points fixes. L'un contient MC et C et l'autre MC et C En effet, par définition de As_A, $M\neg C$ peut appartenir à un point fixe. Cependant pour cela, il est nécessaire que C y appartienne aussi sans quoi MC et C y appartiendront ce qui entrainerait une contradiction.
Or sous S5, on peut déduire $\neg C$ de $M\neg C$ et $MC \rightarrow C$.
Intuitivement la théorie devrait contenir un seul point fixe qui contiendrait MC et C. Sous S5 cela ne sera pas le cas et on remarque que C et $\neg C$ sont placés sur le même pied ce qui ne correspond pas du tout à la sémantique intuitive de l'opérateur M.

La proposition de Mc-Dermott est alors de se restreindre au système S4. Cependant, nous avons vu que ce n'était pas les axiomes de S4 et de S5 qui devraient être remis en cause. En fait, l'axiome qui n'est pas adéquat est $LP \rightarrow P$.
Cet axiome exprime que si l'agent croit P alors P est vrai. Il est vrai que toute théorie stable vérifiera cette formule (puisque si elle contient P elle contient LP et si elle ne contient pas P elle contient $\neg LP$) mais la considérer comme axiome revient à fournir une justification arbitraire à toute croyance. Par conséquent, plus aucune formule de la forme $\neg LP$ ne se retrouvera dans toutes les théories obtenues à partir d'un ensemble initial de premisses. Nous avons vu, par ailleurs, que ces formules étaient à la base de la non-monotonie. Dès lors, inclure l'axiome $LP \rightarrow P$ revient à considérer une logique monotone.

La proposition de Mc-Dermott de se restreindre au système S4 n'est nullement fondée. En effet, l'axiome S5 est tout à fait acceptable mais il met simplement en évidence les conséquences de $LP \rightarrow P$ pris comme axiome. L'axiome S5 peut s'exprimer par $\neg L \neg LP \rightarrow LP$ ce qui permet de croire toute formule P si on ne croit pas sa négation. Dès lors, cet axiome combiné avec $LP \rightarrow P$ nous permet de justifier toute formule.
Le problème réside, en fait, dans la sémantique de l'opérateur L. En logique modale LP signifie "il est nécessaire que P" et à partir de LP il est donc naturel de déduire P. En logique autoépistémique, il signifie "l'agent croit P" ce qui ne nous permet aucunement de déduire P.

Dès lors, si nous voulons baser une logique non-monotone sur une logique modale, c'est le seul axiome qu'il faut supprimer.

Bien que Mc-Dermott ait donné une sémantique formelle à sa logique, il n'a jamais cherché à énoncer ce que représentaient les mondes possibles ou encore la relation d'accessibilité. Autrement dit, bien que la sémantique de l'opérateur M ne corresponde plus exactement à la sémantique qu'on lui accorde dans les logiques modales habituellement, il n'est pas cherché à se représenter ce que signifiaient les axiomes propres de ces logiques. Il n'est guère étonnant dès lors que la sémantique intuitive de l'opérateur M ne soit pas respectée.

3.5 Logique non-monotone et bases de données.

Bien que les logiques non-monotones aient été surtout développées dans le champ "intelligence artificielle", elles ne sont pas sans rapport avec le champ "bases de données". Parmi les problèmes qu'elles abordent, se trouvent les problèmes de représentation des informations négatives ainsi que ceux des informations incomplètes. De plus, elles sont liées au problème de la restauration de cohérence en cas de mise à jour d'une base de données. En effet, dans une base de données conventionnelle, l'ajout d'information ne met jamais en question ni les informations déjà contenues dans la base de données ni les contraintes d'intégrité ce qui traduit un comportement non-monotone. Dans le cas où l'insertion n'est pas cohérente par rapport aux contraintes d'intégrité et aux informations déjà présentes, la priorité sera donnée aux informations déjà présentes et l'insertion sera rejetée pour restaurer la cohérence des informations. Cependant, il est possible d'imaginer que la priorité soit donnée à l'insertion auquel cas, la restauration de la cohérence reviendrait à accepter une exception à une loi générale ou à une contrainte d'intégrité.

exemple : considérons la base de données constituée des deus relations

lieu (cours, local)

an-cours (cours, année)

qui exprime le local où se donne un cours et l'année à laquelle est enseignée un cours.

la règle de déduction

$\forall x$ an-cours (x,"2") $\rightarrow$ lieu (x,"A4")

et la contrainte

$\forall x \forall y \forall z$ lieu $(x,z) \wedge$ lieu $(x,y) \rightarrow z=y$

exprimant respectivement que tous les cours de deuxième année se donne en "A4" et qu'un cours ne peut se donner que dans un local.

Sachant que la base de données contient
"an-cours ("logique","2")" on pourra déduire
"lieu ("logique","A4")
Cependant, l'ajout de "lieu ("logique","S4")"entrainera une incohérence. La restauration de cohérence pourra donner lieu à la modification de la règle générale qui s'exprimera alors

"$\forall x$ an-cours $(x,"2") \wedge x \notin$ "logique" $\rightarrow$ lieu $(x,"A4")$

manifestant un comportement non-monotone.

La prise en compte d'une logique non-monotone peut permettre des restaurations de cohérence où la priorité ne sera pas donnée nécessairement aux informations présentes. De nombreuses relations peuvent également être favorablement représentées au moyen de règles générales, d'exceptions à ces règles et d'exceptions aux exceptions [KOWA 78].
Cependant, peu de recherches ont été effectuées dans ce domaine.
La raison en est que les logiques non-monotones ne disposent pas encore de définitions sémantiques et syntaxiques, suffisamment précises et utilisables.

3.6 Conclusion.

Dans ce chapitre, nous nous sommes efforcés de caractériser les logiques non-monotones.
Nous nous sommes attardés essentiellement sur les logiques autoépistémiques. Les logiques de Doyle et de McDermott ont été présentées. Elles consistaient essentiellement en des modifications de la logique des prédicats et de la logique modale. Nous avons montré leurs défauts qui sont dûs, à notre avis, à la volonté de modifier des logiques existantes qui sont essentiellement monotones. Moore, quant à lui, définit une logique adaptée au traitement de la non-monotonie et ses résultats semblent particulièrement intéressants.
Enfin, nous avons montré, en quoi les logiques non-monotones concernaient le champ "bases de données". Une approche radicalement opposée à celle que nous avons suivie consiste à raisonner dans un espace de théories. En effet, on peut argumenter que la propriété de monotonie ou de non-monotonie est une propriété concernant l'évolution des théories. Pourquoi dès lors tenter de représenter un comportement non-monotone par une nouvelle règle d'inférence qui concerne la déduction au sein d'une théorie ?
Dans cette approche, à partir d'une théorie T qui contient entre autres les axiomes "la plupart des oiseaux volent" et
"a est un oiseau"
de laquelle on ne peut pas déduire que "a ne vole pas", on peut adopter une théorie $T \cup \{\text{vole (a)}\}$. On essaye alors à partir d'une théorie T de déterminer la théorie la plus adéquate à un instant donné.
Il n'est cependant pas évident que cette approche corresponde aux motivations qui ont guidé le développement des logiques non-monotones.
La logique par défaut peut être considérée comme un intermédiaire entre ces deux approches.
La seconde remarque concerne la mise en oeuvre des logiques non-monotones. En effet, à moins que le domaine d'interprétation soit fini ce qui est le cas dans les bases de données, elle risque de poser un sérieux problème. En effet, il n'existe pas de procédure de preuve qui nous disent si une formule quelconque est un théorème ou non.
Tout le problème sera donc de déterminer quand il sera adéquat d'arrêter d'essayer une formule donnée. La sémantique que l'on accorderait à l'opérateur M pourrait ressembler à "dérivable en n étapes n>o".

Enfin, nous concluerons, en remarquant, que les logiques non-monotones,

si elles peuvent contribuer à l'étude des bases de données avec valeurs incomplètes et informations négatives, manquent de bases théoriques concernant leur définitions sémantiques et leur théorie de la preuve.
De manière générale, les travaux concernant la formalisation d'informations incomplètes et d'informations négatives restent fort limités et doivent être considérés comme des points de départ.
Elles sont encore loin de bénéficier d'une sémantique aussi bien définie que les bases de données déductives et leur mise en oeuvre parait assez problématique.

CONCLUSION.

En guise de conclusion, nous rappellerons brièvement les résultats présentés dans ce mémoire.

La contribution de la logique des prédicats du premier ordre peut être étudiée en termes des deux formalisations d'une base de données relationnelle. La formalisation selon la vue "théorie du modèle" est plus adéquate que la formalisation selon la vue "théorie de la preuve" et il n'est guère étonnant que cette formalisation soit la plus utilisée dans le contexte des bases de données. Il est, en effet, plus adéquat d'utiliser cette formalisation pour définir la sémantique d'un langage d'interrogation (représentation des questions) et d'expression des contraintes (représentation des contraintes). De même, cette formalisation est particulièrement adaptée pour montrer l'équivalence entre deux questions (mise en oeuvre des questions) ou pour concevoir et valider une méthode de simplification des contraintes d'intégrité (mise en oeuvre des contraintes). Nous retiendrons surtout le rôle sémantique que peuvent jouer les domaines et l'intérêt des types dans les bases de données. De même, les méthodes de simplification de contraintes d'intégrité semblent être une voie de recherches intéressante.

Bien qu'elle ne soit pas particulièrement adéquate pour la formalisation d'une base de données relationnelle, la formalisation selon la vue "théorie de la preuve" a donné lieu aux bases de données déductives car elle permet de considérer la notion de déduction. Cependant, pour une mise en oeuvre efficace de ces bases de données déductives, il est nécessaire de rendre implicites les hypothèses prises en compte dans les bases de données relationnelles. Pour ce faire, nous avons introduit la règle d'inférence CWA.

La mise en oeuvre des bases de données déductives pose encore de nombreux problèmes d'optimisation et cela quelle que soit l'approche dans laquelle on se place. Les résultats obtenus dans la méthode interprétée nous semblent aujourd'hui les plus avancés et combinent une approche descendante et ascendante. De nombreuses recherches sont effectuées dans ce domaine à l'heure actuelle.

La logique des prédicats du premier ordre a été également utilisée pour les dépendances fonctionnelles et multivaluées ainsi que pour la reconnaissance du langage naturel mais ces résultats n'ont pas été présentés ici.

Cependant, la logique des prédicats du premier ordre présente certaines limitations et nous avons étudié d'autres logiques.

En effet, on peut ne pas vouloir se placer sous l'hypothèse du monde fermé, admettre des informations incomplètes, et représenter des informations négatives explicitement. La logique modale a été étudiée dans le but de voir les possibilités qu'elle offrait pour traiter ce genre de problèmes mais les résultats sont fortement limités.

On peut également ne pas vouloir attribuer une des deux valeurs de vérité à une affirmation et admettre que certaines informations ne sont ni vraies ni fausses mais dépourvues de sens ou encore que leurs valeurs de vérité ne peuvent être déterminées.

Enfin, nous avons considéré les logiques non-monotones qui semblent aborder de nombreux problèmes tels que la représentation des informations négatives, les raisonnements de bon sens et le problème de la restauration de cohérence. Ces logiques semblent être une voie d'avenir même si les recherches restent fort limitées.

Nous rappellerons enfin quelques résultats de la partie préliminaire concernant la programmation en logique. Le contrôle reste une question ouverte. Il sera, en effet, toujours nécessaire mais ses moyens d'expressions sont très divers et aucun consensus n'existe.
Nous avons également émis de sérieuses réserves en ce qui concerne l'équivalence entre les sémantiques déclarative et procédurale. Enfin, nous avons indiqué que la programmation en logique ne supprimerait pas la nécessité d'une spécification. Des recherches dans ce domaine dans la démonstration de programmes logiques, les méthodologies de programmation en logique et les moyens d'expressions du contrôle nous semblent des voies de recherches fort intéressantes.

Nous concluerons en disant que si l'histoire de la logique est fort ancienne, l'interaction logique-base de données est quant à elle très récente et pourrait contribuer au développement des deux champs.

BIBLIOGRAPHIE.

ADIB 83 ADIBA, M. et DELOBEL, C. "Les systèmes relationnels de bases de données" Dunod 83.

ASTR 76 ASTRAHAM et al. "System R : Relational approach to data base management" ACM-TODS vol 1, n°2 (juin 76).

BERN 80 BERNSTEIN et al. "Fast maintenance of semantic integrity assertions using redondant aggregate data" Proc. VLDB conf., Montreal (oct.80).

BLAU 81 BLAUSTEIN, B.T., "Enforcing data base assertion : Techniques and Applications", PL.D. Thesis, Harvard University Cambridge (août 81).

BOYE 72 BOYER R.S., MOORE J.S. "The sharing of structure in theorem proving programs" dans Machine intelligence 7 Edimburgh University Press (72).

BUNE 79 BUNEMAN P., CLEMONS E. "Efficiently monitoring relational Databases", ACM-TODS vol. 4 n°3 (sept 79)

CASA 80 CASANOVA M.A., BERNSTEIN P.A. "The logic of a relational data manipulation language", Proc. ACM Symp. on principles of programming languages, (janv.79) p. 101 - 109.

CHAK 82 CHAKRAVARTY U.S. et al. "Interfacing predicate logic language and relational data bases" Proc. 1 st. conf. on logic programming, Marseille (sept. 82).

CHAN 73 CHANG C.L. et LEE R.C.T. "Symbolic logic and mechanical theorem proving" Academic Press, New-York (73).

CHAN 81 CHANG C.L. "On evaluation of queries containing derived relations in a relational data bases "dans GALL 81 p.201-236.

CLAR 78 CLARK K.L. "Negation as failure" dans GALL 78 p.293-322.

CLAR 79 CLARK K.L., Mc CABE F. "The control facilities of IC-Prolog" Dans Expert systems in the micro-electronic Age" (ED. Mitchie D.) Edimburgh university Press (79).

CLAR 80 CLARK K.L., Mc CABE F. "IC-Prolog language features" dans TARN 80 p. 45-52.

CLAR 82 CLARK et al. "Logic Programming" Academic Press (82).

CLOC 81 CLOCKSIN W.F., MELLISH C.S. "Programming in Prolog" Springer Verlag (81).

COLM 73 COLMERAUER A. et al. "Un système de communication homme-machine en Français. Research Report. Groupe d'intelligence artificielle Université Aix Marseille II (73).

COLM 79 COLMERAUER A. "Un sous-ensemble interessant du Français" RAIRO-revue Française d'Automatique et de Recherche Opérationnelle : Informatique 13,4 (79).

COLM 81 COLMERAUER A., PIQUE J.F. "About natural logic" [GALL 81] p. 343 - 365.

COOD 70 COOD G.F. "A relational model of data for large shared data banks" C.A.C.M. 13,6 (juin 70) p. 377 - 387.

COOD 72 COOD G.F. "Relational completness of data base sublanguage "dans Data base Systems (Rustin.R.ed.) Prentice Hall Englewood Cliffs (72) p. 65 - 98.

COOD 79 COOD E.F. "Extending the relational model to capture more meaning" A.C.M. TODS 4,4 (dec.79) p. 397 - 434.

COOD 82 COOD E.F. "Relational data base : a practical foundation for productivity, C.A.C.M. 25,2 (feb.82) p. 109 - 117.

DALH 79 DALH V."Quantification in a three-valued logic for natural question-answering systems "Proc. 6 th. IJCAI TOKYO JAPAN (août 79).

DALH 82 DALH V. "On data base systems developpement through logic, ACM-TODS. 7,1 (mars 82) p.102 - 123.

DATE 81 DATE C.J. "An introduction to data base systems" (Third edition vol. 1 & vol. 2 . Vol. 1 (1981), vol. 2 (1983).

DAVI 80 DAVID "PROBLEM SOLVING = KNOWLEDGE + STRATEGY" Proc. int. conf. artificial intelligence and information-control systems of robots, Bratislava (juil. 80).

DAVI 80a DAVIS R. , BUCHANAN G.B. "Meta-Level Knowledge : overview and application" Proc. IJCAI (août 80) p. 920 - 927.

DEMO 81 DEMOLOMBE R., NICOLAS J.M. "On the stability of relational queries" dans [GALL 81] .

DEVI 85 DEVILLE Y. "A technique for specifying Prolog procedures" Research paper , FNDP Namur (85).

DINC 80 DINCBAS M. "The metalog problem-solving system, an informal presentation" dans [TARN 80] p. 80 - 91.

EEBI 84 EEBIGLAUS H.D., FLUM J., THOMAS W. "Mathematical logic" Springer verlag (84).

ENDE 72 ENDERTON H.B. "A mathematical introduction to logic", Academy press, New-York (72).

GALL 78 GALLAIRE H., MINKER J. "Logic and data bases", Plenum press, New-York (78).

GALL 81 GALLAIRE H., MINKER J., NICOLAS J.M. "Advances in data base theory", vol. 1, Plenum press, New-York (81).

GALL 81a GALLAIRE H. "A study of Prolog" Automatic program construction" Nato summer school, Gers, (sept. 81).

GALL 82 GALLAIRE H., LASSERRE C. "Meta-Level control for logic programs" dans [CLAR 82].

GALL 84a GALLAIRE H., MINKER J., NICOLAS J.M. "Logic and data bases : a deductive approach" Computing surveys, vol. 16 n°2 (juin 84).

GALL 84b GALLAIRE H., MINKER J., NICOLAS J.M. "Advances in data base theory", vol. 2, Plenum press, New-York (84).

GREE 69 GREEN C. "Theorem proving by resolution as a basis for question-answering systems", Machine Intelligence 4 (Meltzer B. et Mitchie D. eds) American Elsevier Pub. CO., New-York (1969) p. 183 - 205.

HAAC 78 HAACKS. "Philosophy of logics" Cambridge Univrsity press (78).

HAIN 85 HAINAUT J.L., Bases de données, matière approfondie, notes de cours (85).

HAMM 80 HAMMER M.M., ZDONIK S.B. "Knowledge-based query processing" Proc. 6 th VLDB conf. Montreal (oct. 80) p. 137 - 147.

HAYE 73 HAYES P.J. "Computation and deduction" Proc. 2nd MFCS symposium CZEKOSLOVAK academy of sciences (73).

HENS 84 HENSCHEN L.J., NAQUI S.A. "On compiling queries in recursive first order data bases" J.A.C.M. (janv. 84).

HENS 84a HENSCHEN L.J., McCUNE W.W., NAQUI S.A. "Compiling constraint checking programs from first order formulas" dans [GALL 84a].

HEWI 69 HEWI C. "PLANNER : A language for manipulating models and proving theorems in a robot" IJCAI-69 (69).

HILL 74 HILL R. "Lush-resolution and its completness, DCL MEMO n°78, University of Edimburgh (août 74).

HUGH 68 HUGHES G., CRESWELL M. "An introduction to modal logic" Methuen (68).

HOAR 69 HOARE C.A.R. "An axiomatic basis for computer programming" C.A.C.M. 12 (69) p. 576 - 580.

JARK 84 JARKE M., KOCH J. "Query optimisations in data bases systems" Computing suveys, vol.16 n°2 (juin 84).

KENT 83 KENT W. "A simple guide to five normal forms in relational data base theory" C.A.C.M. n° 28 vol. 26 (feb. 83).

KOWA 71 KOWALSKI R.A.,KUEHNER (71) "Linear resolution with selection function" Artificial Intelligence 2(3/4) (71) p. 227-260.

KOWA 74 KOWALSKI R.A. "Predicate logic as programming language" Proc IFIP 74 Congress North Holland (74) p. 569-574.

KOWA 78 KOWALSKI R.A. "Logic for data description" [Gall 78]

KOWA 79 KOWALSKI R.A. "Logic for problem Solving" Artificial Intelligence series (EDS Nilson Nj) North Holland (79)

KOWA 79a KOWALSKI R.A. "Algorithm = Logic + Control" C.A.C.M. 22 (79) p. 424-431.

KING 81 KING J.J. "Quist : a system for semantic query optimizaton in relational data bases" Proc. 7th VLDB conf. Cannes (sept. 81) p. 510-517.

KUHN 67 KUHNS J.L. "Answering questions by computer : A logical Study" Tech Rep. RM-5428-PR, RandCorp Santa Monica (dec.67).

LACR 80 LACROIX M., PIROTTE A. "Associating types with domains of relational data bases Dans "Workoop on data abstraction, data bases and conceptual modeling (Pingree Park, Eolo ACM Sigmod Rec. 11,2 (feb.) p. 144-146.

LECH 82 LECHARLIER B. "Méthodologie de la programmation" Notes de cours (82).

LECH 85 LECHARLIER B. "Réflexions sur le problème de la correction des programmes", Thèse de doctorat, FNDP Namur, (85).

LERO 83 LEROY H. "Théorie de la calculabilité" Notes de cours 83.

LEVE 81 LEVESQUE H.J. "The interaction with incomplete knowledge bases: a formal treatment "IJCAI (81) (Vancouver) p. 240-245.

LLOY 84 LLOYD J.W. "Fondations of logic programming" Springer Verlag, (84).

LOVE 69 LOVELAND D.W. "Mechanical the crem-proving by model elimination"J.A.C.M. vol. 15 n°2 (avril 68) p. 236-251.

LOZI 84 LOZINSKII E.L. "Inference by generating and structuring of deductive data bases", technical report, University of Jerusalem, (84).

MCDE 80 McDERMOTT D., DOYLE J. "Non-monotonic logic I" Artificial Intelligence 13 (79) p. 27-39.

MCDE 82 McDERMOTT D. "Non-monotonic logic II : "Non-monotonic modal theories" J.A.C.M 29 (1) (82) p. 39-57.

MINK 78 MINKER J. "An experimental relational data base system based on logic" dans [GALL 78].

MINK 82 MINKER J. "On indefinite data bases and the closed world assumption" dans "Proceedings of the 6th conference on automated deduction (Springer Verlag) Lecture notes in computer science n° 138 (82) p. 292-308.

MINK 82a MINKER J. NICOLAS J.M. "On recursive axioms in deductive bases. Inf. Syst. 8,1 (jan.82) p. 1-13.

MINS 75 MINSKY M. "A framework for representing Knowledge " Dans "The psychology of computer vision" Winston (ed.) Mc Graw-Hill, New-York (75) p. 211-277.

MOOR 84 MOORE R.C. "Semantic considerations on non-monotonic logic" Artificial Intelligence (84).

NICO 78 NICOLAS J.M., YAZDANIAN "Integrity checking in deductive data bases" dans [Gall 78]

NICO 79 NICOLAS J.M. "Contribution à l'étude théorique des bases de données : apport de la logique mathématique" thèse de doctorat (79).

NICO 83 NICOLAS J.M.,YASDANIAN K. "An outline of B.D.G.E.N. : a deductive DBMS", IFIP (83).

PIRO 77 PIROTTE A. A comprehensive formal query language for a relational data base : FQL "RAIRO Informatique/Computer science vol. 11 n°2 (77).

REIT 78 REITER R. "On closed world data bases" dans Gall 78 .

REIT 78a REITER R. "Deductive question answering on relational data bases" dans [Gall 78].

REIT 80 REITER R. "Equality and Domain closure in first-order data bases" J.A.C.M. 27,2 (avril 80) p. 235-249.

STON 76 STONEBRAKER et al. "The design and implementation of Ingres "ACM-TODS 1,3 (sept 76) p. 189-222.

TARN 80 TARNLUND (EDS)"Proceedings of logic programming workshop", Hongrie (80).

TURN 84 TURNER R. "Logic for Artificial Intelligence" Ellis Horwood Series Artificial Intelligence (84).

ULLM 80 ULLMAN J.D. "Principles of data base Systems" Computer science Press Woodlands Hill (80).

VIEI 85 VIEILLE L. "Handling recusively defined virtual relations in deductive data bases", technical report, E.C.R.C. Munich, (85).

WALK 81 WALKER A.,DROUGH D.R. "Some practical properties of logic programminginterpreters" ESRI ES

WARR 77 WARREN et al. "Prolog : the language and its implementation compared with Lisp". Proc. symp. on A I and programming languages. Siglan Notice 12, n°8.

WARR 81 WARREN D.H.D. "Efficient processing of interactive relational data base queries expressed in logic" dans Proceedings of the 7th VLDB (Cannes 81) p. 272-281.

WIRT 71 WIRTH N. "Program development by stepwise refinement"CACM 14 (avril 71) p. 221-226.

ZLOO 77 ZLOOF M.M. "Query by example : a data base language" IBM Syst. 16,4 (77) 324-343.